Technical Report 918

AFL-1: A Programming Language for Massively Concurrent Computers

Guy Blelloch

MIT Artificial Intelligence Laboratory

This blank page was inserted to preserve pagination.

AFL-1:
A Programming Language for Massively Concurrent Computers

Guy Blelloch

November 1986

ABSTRACT

Computational models are arising in which programs are constructed by specifying large networks of very simple computational devices. Although such models can potentially make use of a massive amount of concurrency, their usefulness as a programming model for the design of complex systems will ultimately be decided by the ease in which such networks can be programmed (constructed). This thesis outlines a language for specifying computational networks. The language (AFL-1) consists of a set of primitives, and a mechanism to group these elements into higher level structures. An implementation of this language runs on the Thinking Machines Corporation, Connection Machine. Two significant examples were programmed in the language, an expert system (CIS), and a planning system (AFPLAN). These systems are explained and analyzed in terms of how they compare with similar systems written in conventional languages.

Revised version of a thesis submitted to the Deparment of Electircal Engineering and Computer Science May 1986 in partial fulfillment of the requirements for the degree of Master of Science. This report describes research done at the Artificial Intelligence Laboratory of the Massachusetts Institue of Technology. Support for the laboratory's artificial intelligence research is provided in part by the Advanced Research Projects Agency of the Department of Defense under Office of Naval Research contract N00014-85-K-0124.

This empty page was substituted for a blank page in the original document.

Contents

List of Figures

List of Tables

Acknowledgments:

I am grateful to Phil Agre and Dave Waltz for many helpful comments and much encouragement. Phil introduced me to the Connection Machine and helped guide much of this research.

I thank Dave Chapman, Tom Knight, John Taft and Dan Weld for looking over various parts of the thesis.

Alan Bawden, Jerry Roylance and Ramin Zabih helped me organize a presentation from which the introduction was derived.

Tom Knight, my advisor, kept me on the practical aspects of the research and guided me away from the slush. Charles Leiserson taught me what I know about computational theory.

I would also like to thank Danny Hillis, Brewster Kahle, Cliff Lasser, Steve Omohundro, Abhiram Ranade and Guy Steele for their help at Thinking Machines.

Chapter 1

Introduction

In recent years there has been much interest in developing computational models for massively concurrent computers. This interest has been motivated by the realization that making effective use of the processing potential available on a thousand or million processor machine is not a trivial problem. Small changes to conventional languages are adequate for the adaptation of highly homogeneous tasks, such as matrix multiplication, but such small changes will not suffice for more general tasks in which communication is not homogeneous and different tasks must run concurrently. In order to implement these tasks, several new programming models have been suggested.

One approach has been to extend conventional symbolic languages such as LISP to include primitives for forking off new processes (nodes in the process graph) and communicating among the nodes [Bawden84, Hewitt83]. In these models, each node and each message is complex. An alternative approach has been to use very fine grained models that have simple nodes, perhaps on the order of logic gates, and send very simple messages. Computations in such models are mostly specified by the connections among the nodes rather than by the computation done at the nodes. This approach can be traced back to McCulloch and Pitts Threshold Logic Elements [1949] and has had a recent resurgence dubbed as "Connectionist Models" [Hinton81, Feldman82, Rumelhart86]. The fine grained models depart more significantly from conventional languages and go further toward accessing concurrency at its roots - at the circuit level.

Although the fine grained models are appealing in many ways, for large applications they suffer three related drawbacks: little work has been done on developing methods to program within the models; the primitives seem too simple to be useful for programming large tasks; and the abstractions supplied by the models differ substantially from those programmers have been raised on, making the models more difficult to work with. This thesis addresses the first two problems by defining a language, AFL-1, for programming the networks and showing how the language allows the user to abstract away from the primitives. The third problem can only be solved with time and experience. To show that it is easy to program in AFL-1, two relatively large tasks were implemented: a production system (CIS) and a planning system (AFPLAN).

To define a language, the thesis first formalizes the notion of "fine grained models with simple nodes" by defining a particular type of network model - the *activity flow network* (AFN) model. The networks AFL-1 creates belong to this class. AFL-1 consists of functions to add nodes and links to a network and a mechanism that allows the user to hierarchically define structure out of these nodes and links. The structure defining mechanism resembles the mechanisms found in some constraint languages [Steele80, Sussman80] and circuit design languages [Batali80].

Although the primitives of AFL-1 are indeed quite simple - no more complex than machine instructions of conventional machines - they are defined independently of any given machine and are therefore - unlike the machine instructions of conventional machines - portable. This machine independence along with the abstraction mechanism makes it possible to program large portable tasks using AFL-1.

Since the modules created in AFL-1 are active networks of integrated code and data rather than blocks of serial code or structures of static data, many of the abstractions and intuitions that have become embedded in the way we think about computation are not useful in the AFN model. For example, the idea of building up complex symbolic structures and manipulating them is not supported by AFNs. This thesis develops some abstractions that are suitable for the AFN model and includes them in the AFL-1 language. One such abstraction is the mutually exclusive group discussed in chapter 7. At present, the AFL-1 support of high level abstractions is weak, but the author hopes that through future development, an environment can be created in which the users can completely remove themselves from the primitives.

To keep the thesis within bounds, the sort of fine grained networks considered is kept simple. In particular the thesis makes the following three restrictions on the networks: a) it does not consider networks that learn, b) it only considers local views in which every node is given a name rather than distributed views such as in [Hinton81, Touretzky85], and c) only considers completely static networks.

The remainder of this chapter gives a short description of activity flow networks, shows an example of how to compute with such networks and program with AFL-1, gives some background on massively concurrent computers, and finally gives a guide to the rest of the thesis.

1.1 AFNs in a Nutshell

An AFN is a static network of simple nodes and links, that:

- only passes unsigned finite approximations of real numbers over its links;

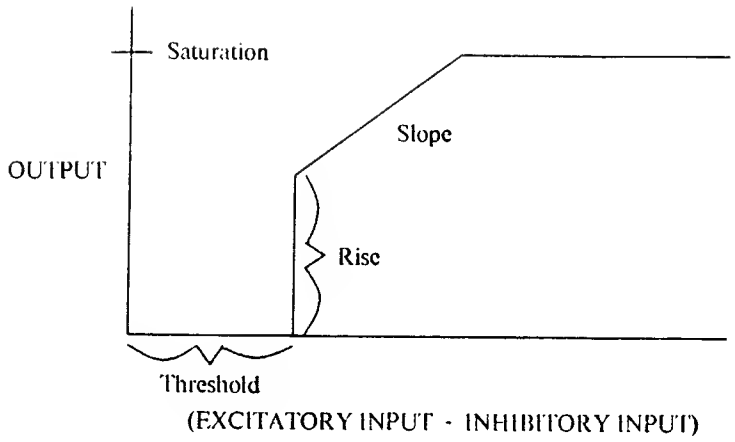


Figure 1.1: The Function of an AFL-1 Node.

- is controlled by a global clock;
- can only communicate to the external world through a set of input and output nodes.

The particular AFN model used by AFL-1 consists of five types of nodes - **input, output, max, min** and **sum**; and two types of links - **inhibitory** and **excitatory**. Nodes can have an arbitrary number of inputs and outputs and have four parameters associated with them - a **threshold**, a **slope**, a **rise** and a **saturation**. The inputs to a node are divided into two types - those from inhibitory links and those from excitatory links. A node combines each input type using maximum, minimum or sum (according to the nodes flavor). Figure 1.1 shows the function applied to these results to get the node's output.

The link makes a directional connection between two nodes. Each link has a weight associated with it. Links multiply their input by their weight and pass this value to their output. The clock is used to synchronize all the nodes after each node sends its message through its output and receives a new message.

1.2 An Example

To give an idea of how one computes with AFNs and how one programs with AFL-1, this section goes through a trivial example. This example is a small subset of the CIS system discussed in chapter 5. To simplify the discussion we will assume that there are only two activity values in the system - inactive (0) and active (1). We will also ignore the issues of input to and output from the network.

```
(make-rule rule-1
  (if has-hair)
  (then is-mammal))

(make-rule rule-2
  (if gives-milk)
  (then is-mammal))

(make-rule rule-3
 (if is-mammal has-hooves)
 (then is-ungulate))
```

Figure 1.2: Three Animal Rules.

The purpose of the AFL-1 language is to build network structure at compile time that can be used to execute some function at run time. To implement a rule based system with such a language one wants a way to translate a set or rules about parameters in the world into a network that can make inferences according to these rules.

To do this, one can make each rule, and each parameter that the rules relate, a collection of nodes and links in the AFN. The rules shown in figure 1.2 might compile into the network shown in figure 1.3.

In this network, if the **asserted** node of a parameter is active it signifies that the corresponding parameter is true. Each **asserted** node takes the Or of its inputs since Max acts as Or for a two valued logic, and each **active** node takes the And of its inputs since Min acts as And for a two valued logic. Because of this, the activation of either the **has-hair asserted** or the **gives-milk asserted** node will activate the **is-mammal asserted** node. Likewise, the activation of both the **is-mammal asserted** and the **has-hooves asserted** nodes will activate the **is-ungulate asserted** node. We assume the **gives-milk, has-hair and has-hooves asserted** nodes are activated either through other rules or from **input** nodes.

The system uses the **want-to-know** nodes for goal directed reasoning. In goal directed reasoning if the system wants to know the value of a parameter, it traces back through the rules from the **then-part** to the **if-parts**. The backward links in figure 1.3 have this effect. For example, if **has-hooves** is not known, then the **want-to-know** node of **is-ungulate** will activate the **want-to-know** node of **has-hooves**.

To get from the rules defined in figure 1.2 to the network shown in figure 1.3 one can define a generic rule structure as shown in figure 1.4A and a generic parameter structure as shown in figure 1.4B. These structures can be instantiated once for each rule and parameter in the rule set. AFL-1 calls these generic structures **groups** and supplies the **defgroup** form for defining them. Figure 1.5 shows what the definitions of the parameter and rule groups might look like.

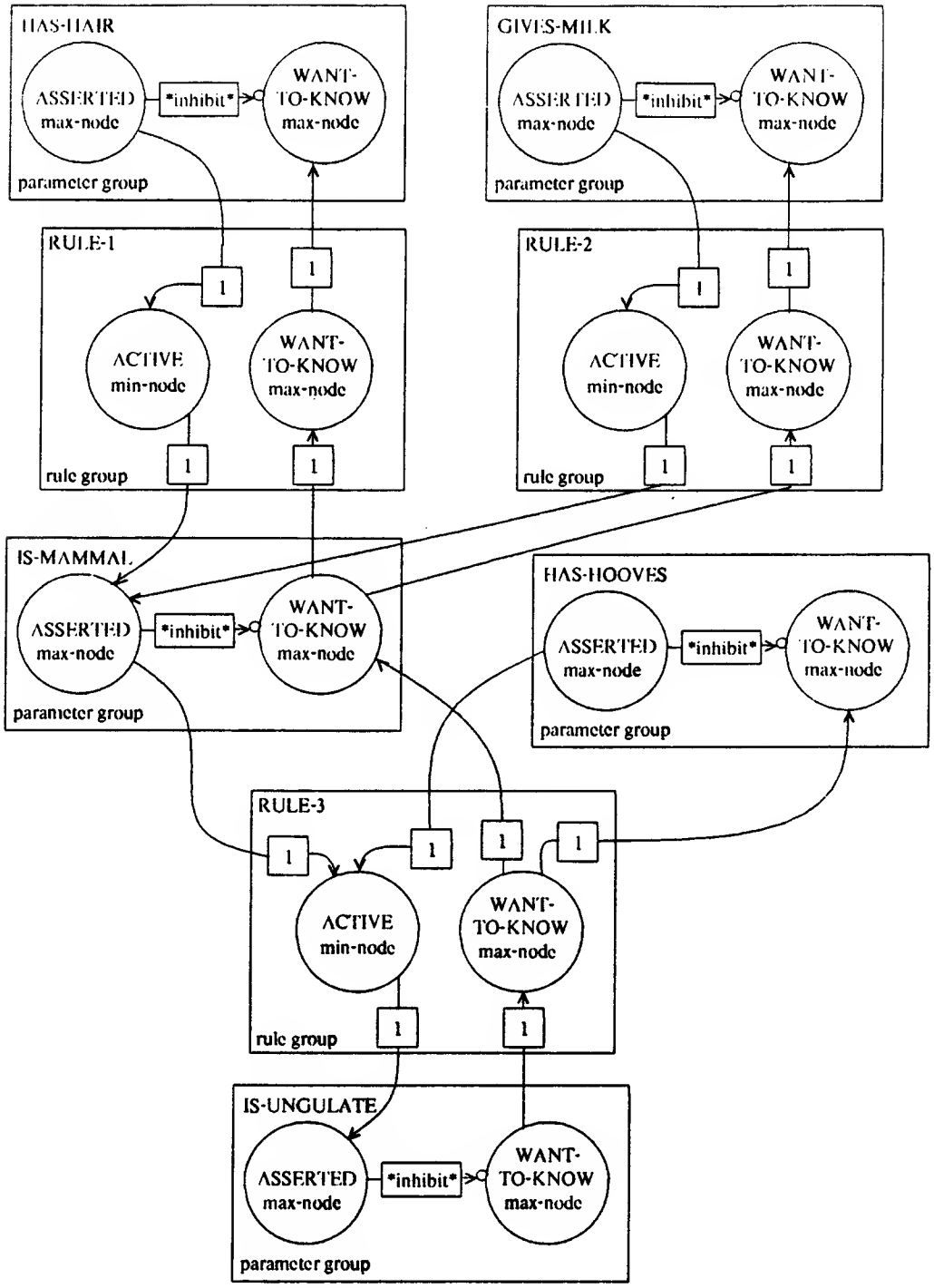


Figure 1.3: An Example Network for Three Rules.

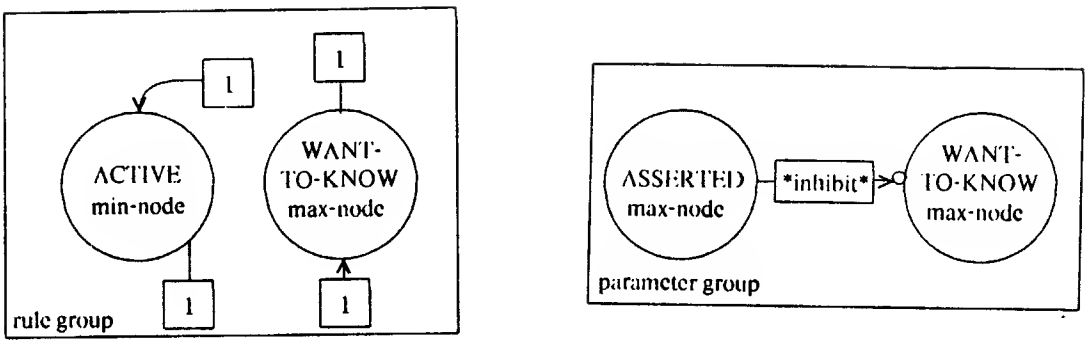


Figure 1.4: The Parameter and Rule Groups.

```
(defgroup parameter ()
  (make-max-node 'asserted)
  (make-max-node 'want-to-know)
  (make-inhibitory-link 'asserted 'want-to-know))

(defgroup rule (if-parts then-parts)
  (make-min-node 'active)
  (make-max-node 'want-to-know)
  (dolist (if-part if-parts)
    (make-excitatory-link '(< ,if-part asserted) 'active)
    (make-excitatory-link 'want-to-know '(< ,if-part want-to-know)))
  (dolist (then-part then-parts)
    (make-excitatory-link '(< ,then-part want-to-know) 'want-to-know)
    (make-excitatory-link 'active '(< ,then-part asserted))))
```

Figure 1.5: The Definition of the Parameter and Rule Groups.

Although not shown in the example, the definition of a **group** can include another **group** within it. This allows a hierarchical definition of complex **groups**.

1.3 Massively Concurrent Machines

Researchers have proposed many concurrent computer architectures each of which is more appropriate for implementing some models of concurrent computation than others. Two important aspects of these architectures are the granularity of the processors, and the interconnection scheme among the processors. The granularity will tell us whether the architecture is single instruction multiple data (SIMD) or multiple instruction multiple data (MIMD) and whether it is best at double precision floating point multiplies or manipulating a small number of bits. The interconnection scheme will tell us how efficiently the architecture can simulate the connections used by various models. Table 1.1 classifies some of the existing concurrent architectures by their granularity and interconnection scheme.

Activity flow networks are well suited for fine grained architectures with a general interconnection scheme. They are well suited for a general connection scheme because they make no restrictions on the connections among nodes. They are well suited for fine grained architectures because the nodes only manipulate small numbers of bits and networks consist of a large number of nodes. The only existing architecture that is fine grained and has a general connection scheme is the Connection Machine (CM). The functions executed by the nodes of an AFNs are actually significantly simpler than what the CM processors are capable of executing, so an even finer grained machine might be better suited.

1.3.1 The Connection Machine

Figure 1.6 shows the schematic block diagram of the existing Thinking Machines Corporation, Connection Machine. This machine has 2^{16} processors. Each processor is bit serial and has on the order of 10^4 bits of local memory. The machine only has a single instruction stream that is broadcast from the microcontroller. The router of the CM allows any processor to send a message to any other processor. A message can either be sent by having the address of the other processor and dynamically routing it, or by setting up a static path to the other processor at compile time. A host machine controls the connection machine. Currently a Symbolics 3600 is used as the host.

This thesis is motivated by the existence of a CM and by related machines being designed. Without a Connection Machine, running networks created by AFL-1 would

Granularity		
Fine Grain (SIMD) ($10^4 \rightarrow 10^6$ procs)	Medium Grain ($10^2 \rightarrow 10^4$ procs)	Coarse Grain ($< 10^2$ procs)
Connection Machine [Hillis85]	Dado [Stolfo83]	C.mmp [Siewiorek78]
MPP [Batcher80]	Ultracomputer [Gottlieb83]	Illiac-IV [Bouknight72]
Staran [Batcher74]	BBN-Butterfly [Crowther85]	S-1 [Farmwald84]
GAPP [Davis84]	PEPE [Thurber76]	HEP [Smith84]
	Cm* [Siewiorek78]	
	Non-Von [Shaw85]	

Connection Scheme				
General		Barrel Shift	Tree	Grid
High Bandwidth	Low Bandwidth			
Connection Machine	Cm*	Staran	Dado	MPP
Ultracomputer			Non-Von	Illiac-IV
BBN-Butterfly				GAPP
S-1				

Table 1.1: Classification of Concurrent Computer Architectures.

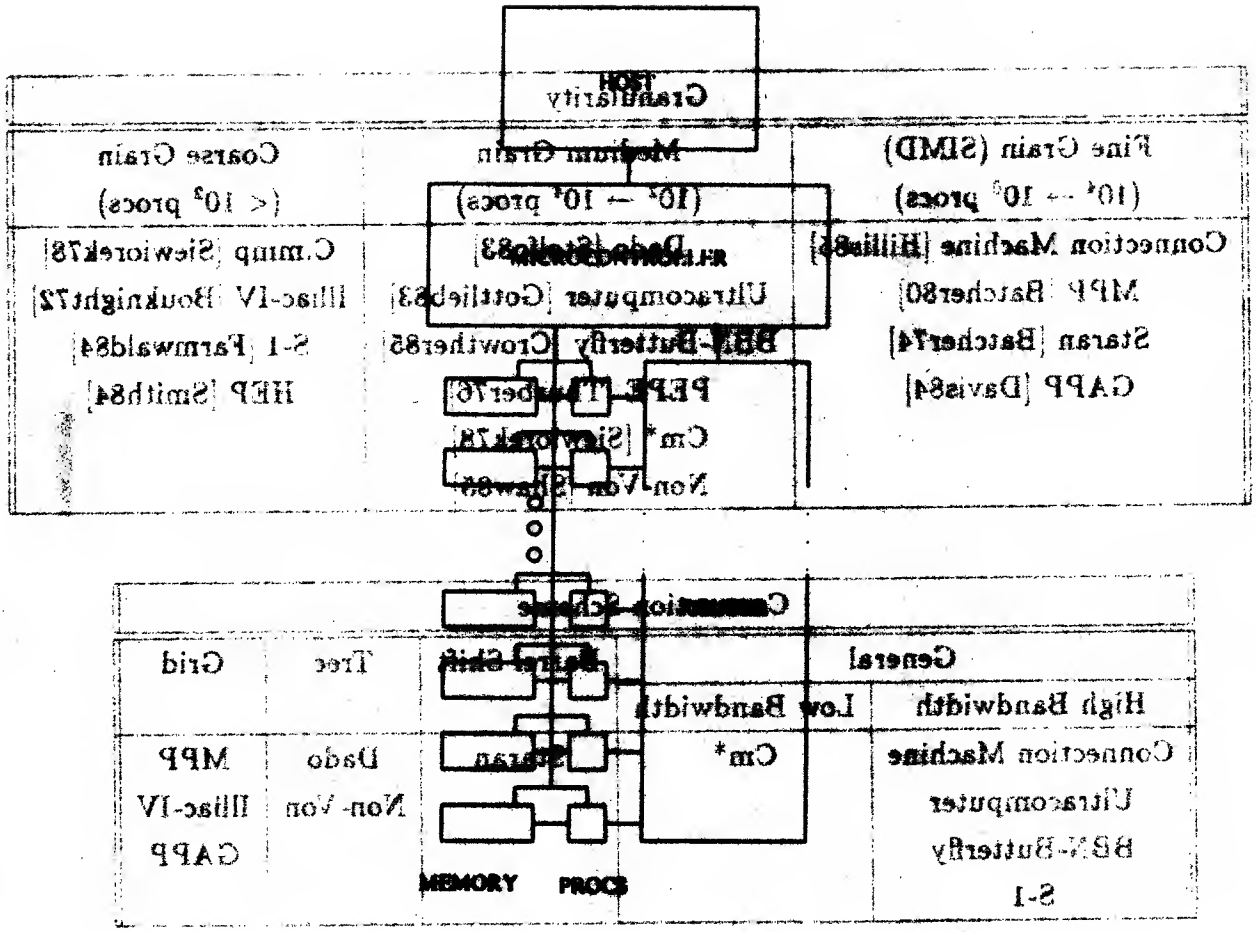


Figure 1.6: Block Diagram of Connection Machine.

be impractical.

1.3.2 Taking Advantage of SIMD

Every processor in the CM has a context flag and each processor only listens to the instruction stream if its context flag is set. Using this flag it is possible for the CM to simulate a multiple instruction multiple data (MIMD) machine by serially cycling through all the instructions needed by different types of processors. This cycling is called time multiplexing the instruction stream.

For example if there are two types of processors in the machine, type A and type B, then the instruction unit can broadcast the instructions for the type A processors while the type B processors sit idle and then broadcast the instructions for the type B processors while the type A processor sit idle.

Instruction multiplexing is useful but costs time proportional to the number of processor types. By having a model with a minimal set of primitive elements, the idle time of the processors can be kept low. Since AFL-1 only creates a few primitive elements, little time is wasted multiplexing the instruction stream. In fact, since the majority of the time is taken sending messages and all the primitives can send their output values together, almost no time is wasted multiplexing the instruction stream.

1.3.3 Implementation

The implementation of AFNs on the CM uses a separate processor for each link as well as each node. Because of this, when evaluating the size of a network, the links should be counted as well as the nodes.

It is desirable not to have machine dependent restrictions on the size of a network. On the CM, the implementation can include more links and nodes than processors by placing multiple virtual processors (VPs) on each physical processor. By doing this it is reasonable to implement networks with several million links. The time taken by each step of the network goes up slightly more than linearly with the number of VPs per processor.

On the 64K CM when running a network with a million links (16 VPs/proc), a step of the AFN takes about 15 mili-seconds. Chapter 8 explains the implementation.

1.4 Thesis Organization

There are several system levels discussed in this thesis. An attempt was made to make a clean separation among the implementations of these systems. This section outlines the

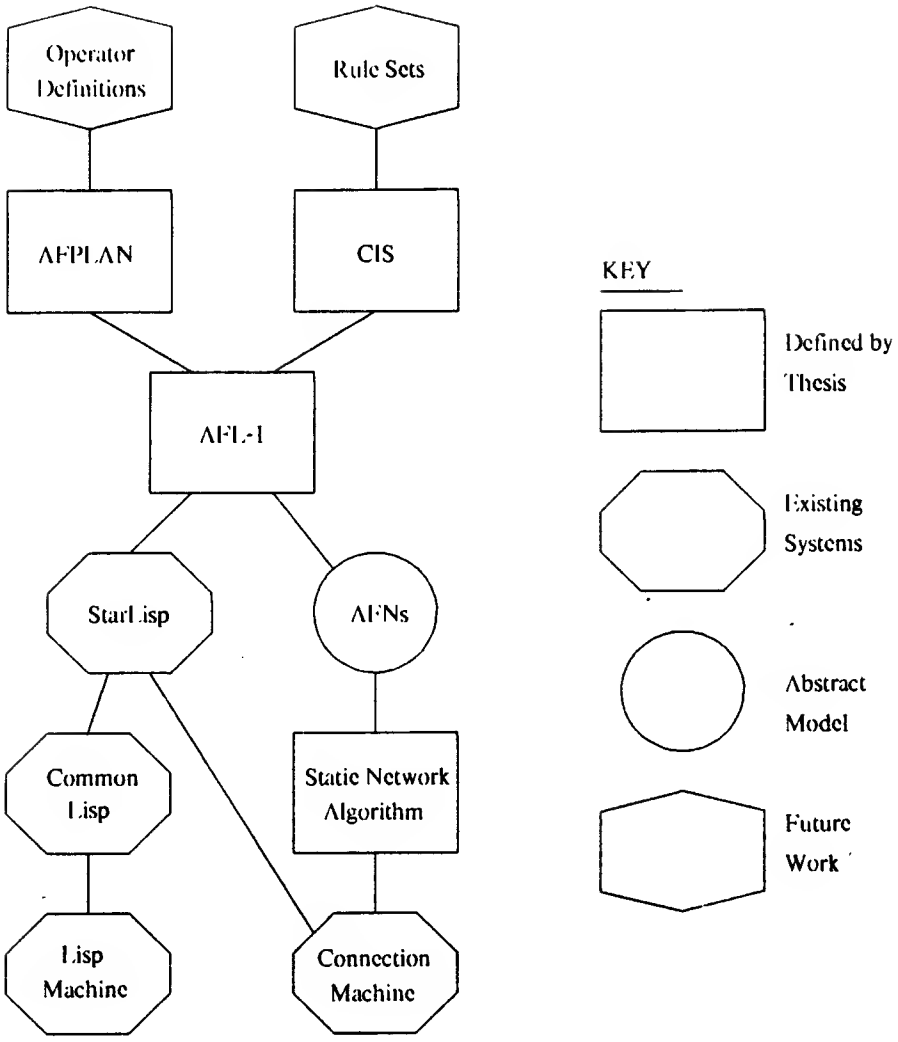


Figure 1.7: Levels of Systems Discussed in this Thesis.

systems and gives pointers to the sections or chapters where the systems are discussed. Figure 1.7 shows how the systems fit together.

The Connection Machine architecture was outlined in section 1.3.1. A more complete description can be found in [Christman84, Hillis85]. The CM uses a Symbolics 3600 as a host [Symbolics85]. All the software is implemented as an extension to Symbolics Common Lisp [Steele84, Symbolics85]. StarLisp is a language developed at Thinking Machines Corporation for programming the connection machine [Lasser86].

The static network algorithm is an algorithm developed for running static networks such as logic simulators, circuit simulators or NETL [Fahlman79] like systems on the CM. Static networks are discussed in section 2.1. The algorithm is discussed in section 8.3.

The activity flow network model is a particular static network model. The AFN model was outlined in section 1.1. How the model relates to other network models of computation is discussed in chapter 2.

AFL-1 is a language for creating AFNs. It is an extension of the Symbolics Lisp environment and StarLisp and uses the static network algorithm on the CM to run the networks it creates. The language is discussed in chapters 3, 4 and 7.

To show how to program with AFL-1. two significantly large tasks were implemented with the language. Chapter 5 describes the concurrent inference system (CIS), a production system and Chapter 6 describes AFPLAN, a planning system. On top of these systems one can implement rule sets or planning worlds for a particular domain. This thesis only gives small examples of such rule sets or planning worlds. These examples are included in the discussion of the systems.

Chapter 2

A Hierarchy of Network Models

To show how activity flow networks (AFNs) relate to other models of computation, this chapter outlines a hierarchy of network models of concurrent computation. As Fahlman noted [Fahlman83], a good way to categorize such models is by the complexity and content of the messages. Such a measure is useful because it indicates much about the model in general, such as the complexity of the nodes and the efficiency of the implementation.

Figure 2.1 shows a hierarchy of models categorized by their messages. Like any categorization, the boundaries between the models are not perfectly sharp - many models slip in between the categories. This chapter starts at the top of this hierarchy and works down to the activity flow model. Branches that lead to activity flow networks are explored in more detail than other branches.

2.1 Static and Dynamic Networks

The class of all network models of concurrent computation can be broken into two subclasses: static and dynamic networks. In a static network each node communicates with a fixed set of other nodes, while in a dynamic network each node can dynamically choose what other nodes it wants to talk to.

In practice static networks are compiled and loaded into the hardware that is going to run them. Communications tend to be quicker on compiled static networks than on dynamic networks for three reasons. First, algorithms can be used when the network is compiled to place nodes that communicate with each other physically close so that long distance links are minimized. Second, since the switches within the routing network are only set once, much time can be spent determining good ways of utilizing the bandwidth of the routing network. Third, no address has to be sent, and the routing-switches do not have to be set dynamically. For examples, in logic simulators the address can be much longer than the data and therefore if routed dynamically requires a large portion of the routing time.

Dynamic network models include CL1 [Bawden84a], CGL [Bawden84b] and Apiary [Hewitt83]. These models all allow passing of pointers (connections in the case of CGL)

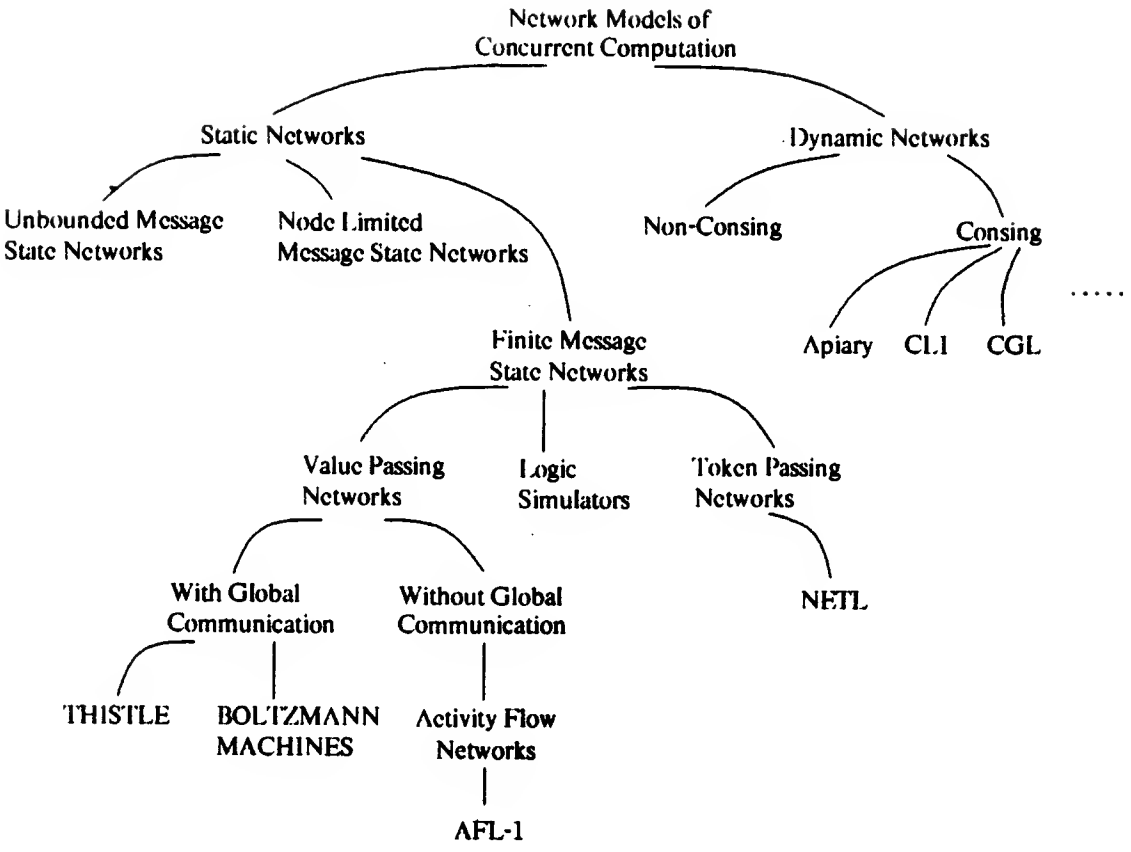


Figure 2.1: A Taxonomy of Concurrent Network Models.

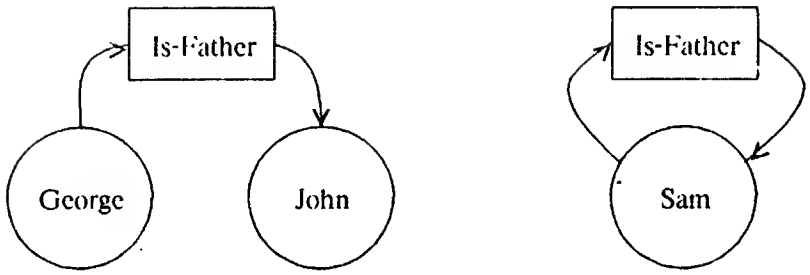


Figure 2.2: Example Network for NETL.

that reference particular nodes, among the nodes in the process graph. They then allow a node to communicate directly with any node it has a pointer or connection to. Such a model requires dynamic routing since the set of nodes a particular node communicates with dynamically changes over time.

All the dynamic networks the author knows about also allow dynamic **consing** of new nodes in the process graph. **Consing** is the process of creating new nodes. The CGL system computes almost entirely by splitting nodes into collections of nodes and merging pairs of nodes into a single node.

2.2 Static Networks (Message Complexity)

Static networks can be divided according to the complexity of their messages into three groups: finite message state networks, node limited message state networks and unlimited message state networks. Finite message state (FMS) networks only send one of a finite set of states over the links. The size of this set of states must be independent of the number of nodes in the network. Node limited message state (NLMS) networks send messages with a number of states proportional to the number of nodes in the network - this allows pointers to be passed. Unlimited message state (UMS) networks can send arbitrarily complex messages.

Since the size of a message is a function of the number of states (the logarithm), the FMS networks have fixed size messages. In practice the messages of an FMS tend to be short. All FMS networks are static since for large networks they do not have enough state in their message to encode pointers. Pointers grow with the number of nodes in the network.

With NLMS networks it is possible to send pointers since the number of states of the messages can grow with the size of the network. An example of a static NLMS network, is the *painted marker* networks suggested in [Fahlman81]. Although these

networks are static, they can send pointers (painted markers) among the nodes so that each node can identify the original source of the message. This is useful in determining whether two messages came from the same place. For example, in semantic networks of the sort used in NETL [Fahlman79] pointers can be used to answer questions such as "Find all persons who are their own father" for the network shown in figure 2.2. This question can not be answered with FMS networks.

UMS networks do not exist in practice since it is hard to support unbounded message lengths, but they might be of theoretical interest. The general definition of Data Flow networks are UMS networks but in practice only finite fixed length messages are allowed.

The implementation of FMS networks is the easiest and most efficient. Since in FMS networks the length of the messages are known, usually short, and equivalent across the network it is very easy to route them. Also with FMS networks it is usually easy to combine messages as they converge at a single node. Section 8.3 describes an algorithm that can be used with minor changes on the Connection Machine to implement most FMS networks.

2.3 FMS Networks (Tokens, Values or Logic States)

Within the class of finite message state (FMS) networks we can further categorize the networks by the objects that the messages represent. Three categories are token passing networks such as in [Collins75], [Fahlman79] and [Woods78], value passing networks such as most of the connectionist models [Rosenblatt61, Hinton81, Feldman82, Rumelhart86] and logic simulators such as [Denneau82]. Although the object that the message represents does not make a great difference to the implementation, it affects what the models are used for.

Token passing networks only pass tokens along the links. Each message consists of a single token. The set of tokens (possible states of the message) have no order defined over them. Value passing networks (VPNs) only pass finite approximation of the real numbers over the links. Such representations could be as simple as a 4 bit integer or as complex as double precision floating point numbers. They differ from token passing networks in that there is an order defined over the possible states of the message.

Logic simulators pass logic levels over the links. The number of states this level can have varies from simulator to simulator but typically consists of 2 through 8 states - possibly 1, 0, Uninitialized, Float-High, Float-low, and Error. Switch level simulators [Hayes82] and circuit level simulators [Deutsch84] are also FMS networks.

2.4 VPNs (Global Communications)

We can further categorize VPNs by whether they allow global communications. Global communications include such things as taking a global OR of some value in all the nodes and redistributing this value back to the nodes. Such global communications allows for a secondary, often implicit, means for nodes to communicate.

In Thistle, a VPN suggested by Fahlman [1983], global communications are used heavily. For example, the host might put a value on all the **human** nodes and then have those nodes send the value over their **hair-color** links. The system might then take a global OR of the **brown** nodes to see if any brown-haired humans exist.

In contrast, activity flow networks (AFNs) do not allow any global communications. The only global control in AFNs is a clock signal that allows the network to take a step. Since the clock signal does not give a path from the nodes back to the controller, it does does not allow for any communications among nodes. AFNs are also defined to only send unsigned values (activities).

Many connectionist models are AFNs but some are not strictly AFNs because of global control, such as those in [Ackley85, Touretzky85], or because some messages are not finite representations of real numbers, such as in [Feldman82].

Chapter 3

AFL-1

AFL-1 is an experimental programming environment in which one can construct, run, examine and debug Activity Flow Networks (AFNs). In large measure, AFL-1 was designed to investigate the plausibility of programming large AFNs, and to suggest directions for future work in related languages. The AFL-1 programming environment conceptually consists of two pieces of hardware, a **host-machine**, which is used to construct, examine and debug AFNs, and a **network-processor**, which is used to run the networks. The part of the environment concerned with constructing AFNs will henceforth be called the "AFL-1 language". This chapter and the next describe the AFL-1 language and chapter 8 describes the **network-processor**. The thesis includes little discussion on examining and debugging networks.

As with conventional languages, the AFL-1 language supplies a set of primitives and a mechanism to abstract away from these primitives. The primitives define an abstract machine on which networks are built. The abstraction mechanism allows the programmer to group the primitives so that networks can be more easily repeated, debugged, examined and described. To capture common programming cliches, AFL-1 supplies a set of predefined abstractions, and includes libraries of application dependent abstractions.

Previous languages, such as INSCON [Small82], designed for constructing connectionist style networks have only supplied forms for creating the primitive elements. AFL-1 takes some ideas from these languages and some abstraction ideas from work on constraints and circuit design [Sussman81, Steele80, Batali80], to create an environment for programming large AFNs.

This chapter gives a formal definition of the primitives and discusses why they were selected. The outline given in section 1.1 is probably adequate for the first pass over this thesis. In AFL-1, one builds levels of abstraction with **node groups**. A **node group** defines a collection of primitives (nodes and links) that can be instantiated within an AFN. Chapter 4 discusses **node groups** and how they are defined and used.

The AFL-1 environment is an extension to the Symbolics LISP environment and currently uses as the **network-processor** either the Connection Machine or a simulator running directly on a Lisp Machine. All the forms discussed in this chapter and most

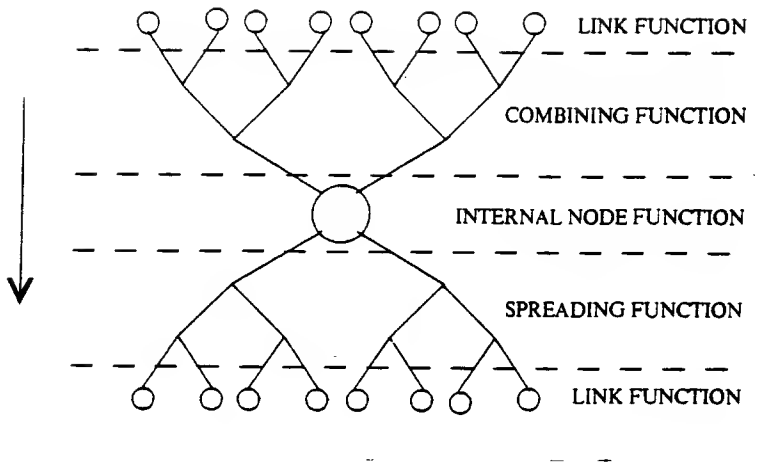


Figure 3.1: The Functions Performed by an AFN.

of the forms discussed in chapter 4 are included in these implementations. Since the AFL-1 environment is relatively large and is still changing, this thesis only introduces the ideas of the language and is not meant as a manual.

Although connectionist networks can potentially make use of a massive amount of concurrency, their usefulness as a programming model for the design of complex systems will ultimately be decided by the ease in which such networks can be programmed (constructed).

3.1 Properties Of The Primitives

As with all languages, it is important that the AFL-1 primitives are cheap to implement on the class of machines (**network-processors**) the language is intended for, but that they do not depend on hardware which is specific to a single machine; and that the primitives are useful for programming the types of applications the language is intended for, but not designed for a particular application. The primitives should supply a clean and portable interface between Activity Flow Programs and the machine they run on. These considerations guided the design of the AFL-1 primitives.

To make the primitives intuitive, two types of elements are supplied, a **node** and a **link**. The **link** element is an active element, not just a passive wire, and has a single input and single output. The **node** is an active element and can have an arbitrary number of inputs and outputs. One should note that functionally equivalent networks can be constructed with a single type of element that only has two inputs and outputs. AFL-1 does not use a single element type since it seems that this would make constructing networks less intuitive.

Given that the network is divided into nodes and links, the work done by the **network-processor** on the Activity Flow Network can be divided into four parts (see Figure 3.1):

- Combining function - combines the activities from many link elements into a single activity and passes this activity to a node element.
- Internal node function - a function performed on the internal parameters of each node and the activity received from the combining function. The internal parameters are set at compile time.
- Spreading function - takes the result of the internal node function and spreads it to all the output links of the node. This can be implemented on a fan-out tree.
- Internal link function - a function of the internal parameters of a link and the activity received from the spreading function. Link functions are usually simpler than node functions.

Since the spreading and combining functions get executed more often than the node and link functions, they should run faster.

The following 5 sections describe the primitives in detail.

3.2 AFL-1 Nodes

Each node in AFL-1 has two sets of inputs, an inhibitory-set and an excitatory-set, and a single set of outputs. Each of these sets can have an arbitrary number of elements (links). All that can be passed through an input or output is an activity value (defined in section 3.6). There are three different types of nodes: **max, min** and **sum**. These types only differ in the combining function performed on their input sets. Each node has four parameters that are set when the node is created at compile time. These parameters are the threshold (T), slope (S), rise (R), and saturation (Sat).

At run time all nodes are controlled by a global clock. On the clock signal they perform the following routine in lock-step.

```
CASE Node-Type OF
  Sum : ivalue := SUM(excitatory-inputs) - SUM(inhibitory-inputs)
  Max : ivalue := MAX(excitatory-inputs) - MAX(inhibitory-inputs)
  Min : ivalue := MIN(excitatory-inputs) - MIN(inhibitory-inputs)
IF (ivalue < Threshold)
```

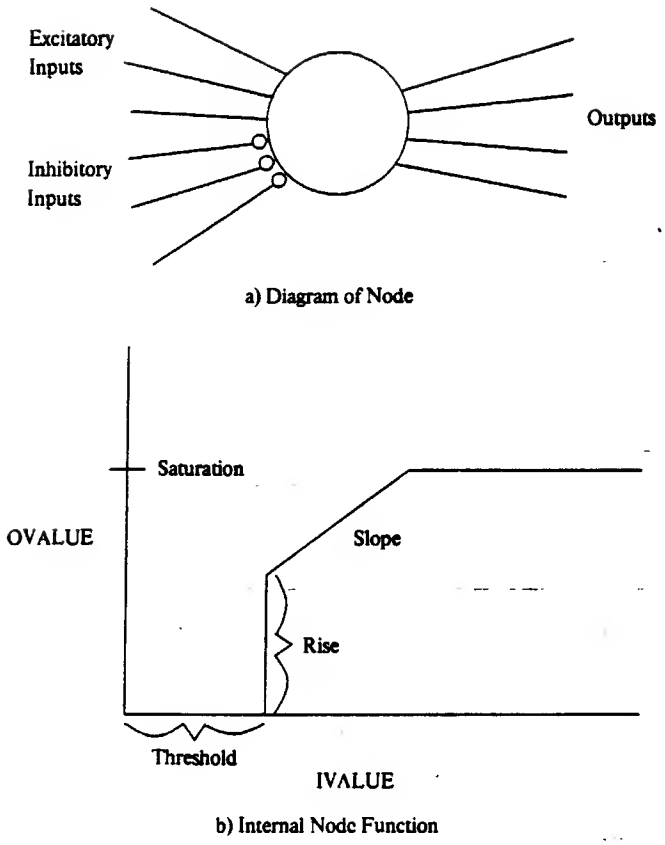


Figure 3.2: The AFL-1 Node.

```
    THEN  ovalue := 0
    ELSE  ovalue := MIN((ivalue * Slope) + Rise, Saturation)
outputs := COPY ovalue
```

The formal definitions of the operations $(+, -, *, <, MAX, MIN, SUM, COPY)$ over the set of possible activity values, is given in section 3.6. The internal node function, the part of the above routine that maps **ivalue** to **ovalue**, is piecewise linear and is shown in figure 3.2.

To add a node primitive to the activity flow network, one of the functions **make-sum-node**, **make-min-node** or **make-max-node** is used. These functions have the following form.

```
(MAKE-SUM-NODE node-name &optional (threshold *active*)
                                   (slope *nil*)
                                   (rise *active*)
                                   (saturation *saturation*))
```

The **make-max-node** and **make-min-node** functions require the same arguments. **Make-sum-node** can be abbreviated with **make-node**.

3.3 AFL-1 Links

There are two kinds of links in AFNs, inhibitory and excitatory links. An inhibitory link connects the output of one node to the inhibitory input of another node, and the excitatory link, connects the output of one node to the excitatory input of another node. Each link has a single input and output and a single parameter, its weight (W). This parameter is set at compile time. At run time the links multiply their **input** by their **weight** and output the result.

```
ovalue <- ivalue * WEIGHT
```

The **make-excitatory-link** and **make-inhibitory-link** functions add links to the AFN.

```
(MAKE-EXCITATORY-LINK from-node to-node &optional (weight 1))
(MAKE-INHIBITORY-LINK from-node to-node &optional (weight 1))
```

3.4 Input And Output

In addition to **sum**, **min** and **max** nodes, AFL-1 includes the **input** and **output** node primitives. These nodes are connected to the "outside world" and are the I/O of an activity flow network. The outside world is anything outside of the AFN that activates input nodes and is affected by output nodes; for example, a terminal, a bank of sensors or the motor controls for a robot.

In most programs discussed in this thesis, the **input** and **output** nodes are connected to an external serial computer which interprets the output nodes and sets the input nodes: the serial computer acts as an interface between the user and the AFN. Since the interaction with the user is necessarily sequential, the serial computer does not act as a bottleneck for such communications.

The **make-input-node** and **make-output-node** functions add **input** and **output** nodes to the AFN. The output node type uses the **sum** combining function.

```
(MAKE-INPUT-NODE node-name)
(MAKE-OUTPUT-NODE node-name &optional (Threshold *active*)
                                      (Slope *nil*)
```

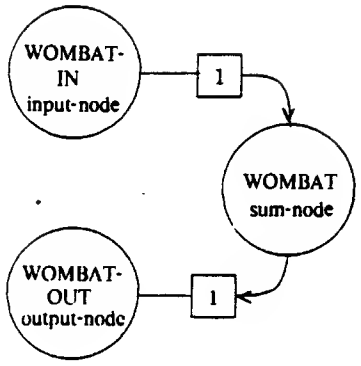


Figure 3.3: A Simple AFL-1 Network.

```
(Rise *active*)
(Saturation *saturation*))
```

Figure 3.3 shows a simple network of nodes and links.

3.5 The Afl-Step

Once the programmer constructs an AFN out of the primitive elements, she runs the **network processor** by executing **afl-steps**. An **afl-step** consists of one cycle through the four functions of the **network-processor** - the spreading function, the link function, the combining function and the node function. The step is used to synchronize all the processors and serves as the basic time unit of the **network-processor**. The implementation is free to execute each of the substeps asynchronously but at the end of the whole step all the processors must have received all their inputs and have generated a new output. A computation will consist of many **afl-steps**.

Chapter 8 discusses how the **afl-step** is implemented.

3.6 The Activity Values

Although the activity values are best imagined as real numbered values, implementations require a finite representation. For efficiency, it is best if a low-precision fixed point representation is used. With fixed point numbers, many of the operations can be done bit serially in a single pass. AFL-1 uses unsigned fixed point numbers with 3 bits on each side of the point to represent activities. The set of possible activity values will henceforth be called the "activity set" and is shown in figure 3.4.

All objects passed around in the AFN at run time, and all the parameters of the nodes and links set at compile time, belong to the activity-set. The operations

```
activity-set = (0, 1/8, .... , 61/8, 62/8, *saturation*)

The following elements are equivalent:
        0 and *nil*
        1 and *active*
    63/8 and *saturation*
```

Figure 3.4: The AFL-1 Activity Set.

$(+, -, *, <, MAX, MIN, SUM, COPY)$ are closed over the set and are defined as follows.

The operation "+" adds two elements of the activity-set and returns the result. If the addition overflows, the element *saturation* is returned. The operation "−" subtracts two elements of the activity-set and returns the result. If the subtraction underflows, the element *nil* is returned. The operation "$*$" does fixed point multiplication on two elements of the activity-set. If the multiplication overflows, the element *saturation* is returned. Since the multiplication creates 6 bits below the point, the lowest three bits are dropped.

The operations **binary-min**(A, B) and **binary-max**(A, B) for A and B elements of the activity-set, return the lower and higher valued activity respectively. The relative values are determined by regular fixed point ordering.

$SUM(V1, V2, ..., VN)$, $MIN(V1, V2, ..., VN)$, and $MAX(V1, V2, ..., VN)$ for all V in the activity-set are defined as the value of the root of a binary tree which has the elements Vi at the leaves, and where +, **binary-min**, and **binary-max** respectively are applied at all the nodes. Since the the three operations +, **binary-min**, and **binary-max** are associative and commutative, the three combining functions are invariant under the construction of the fan tree.

The operation $PLACES < -COPY(A)$, for A in the activity set, and for $PLACES$ a set of slots which can accept a single element of the activity set, takes the element A and copies it into all the slots specified by $PLACES$.

The operation (A < B) is true if A appears before B in the activity-set, and is false otherwise.

3.7 Design Decisions

This section gives a short discussion of why the AFL-1 primitives where selected.

3.7.1 The Combining Functions

Like the Thistle Machine suggested by Fahlman [Fahlman82, Fahlman83], AFL-1 supplies three combining functions, *MIN*, *MAX* and *SUM*. Each of these functions is commutative, associative and and with fixed point numbers can be executed bit serially in a single pass. This allows these functions to be implemented efficiently. The only other well known functions with these properties are bitwise *AND* and *OR* but since these assume a bitwise representation, they are not included in AFL-1.

It is possible to implement the **min** and **max** nodes with just **sum** nodes but this entails a significant cost. Since the **min** and **max** nodes are generally useful for programming and cheap to implement directly, AFL-1 supplies them as primitives. **Min** and **max** nodes can be used to efficiently implement Zadeh's rules of fuzzy reasoning (see section 5.9), and they allow efficient implementations of Mutually-Exclusive groups (see section 7.1.6). **Min** and **max** are cheap to implement on digital computers but are not necessarily cheap to implement on analog computers.

A problem with the *SUM* combining function is that it will easily saturate with large fan-ins.

Section 3.8 discusses some additional combining functions that might be reasonable to add to AFL-1.

3.7.2 Inhibitory and Excitatory Inputs

To make the combining functions tree invariant (associative and commutative), but to still allow inhibitory links, separate inputs have to be used for the inhibitory and excitatory links. If a single set of inputs is used, and negative weights are allowed on the links, the + operation is not associative. For example, for saturations of $-63/8$ and $63/8$, the expression $(60/8 + 30/8) + -60/8$ is equal to $3/8$ while $60/8 + (30/8 + -60/8)$ is equal to $30/8$. Since an arbitrarily large fan-in is allowed, no *finite* saturation would solve this problem. If the combining functions are not tree invariant, the function of a network depends on the order in which links are added to a node.

3.7.3 Analog Output

Many Neural Networks and Connectionist models use digital instead of analog outputs [McCulloch45, Rosenblatt62, Hinton85a]: here analog is meant as a signal with more

than two levels. In these models, the spreading function manipulates digital values while the internal link, combining and internal node functions manipulate analog values.

AFL-1 uses analog values (activities) throughout the network. The analog values allow more state to flow among the nodes in the network and have little implementational cost.

3.7.4 Piecewise Linear

Because linear functions are cheap to implement and easy to interpret, all the functions performed by AFL-1 are piecewise linear. On bit serial machines it requires significantly less time to compute linear functions than exponential functions, or polynomial functions of order greater than 1. It is also easier for a programmer to understand the effect of the parameters Threshold, Slope, Rise, Saturation and Weight, than the coefficients of a polynomial function.

Some researchers have suggested that the **sigmoid** function has good properties for neural like elements [Rumelhart86]. It is possible to categorize the **sigmoid** function by a set of parameters that approximate a Threshold, Slope and Saturation thus allow an intuitive description. It is also possible to approximate a sigmoid function with a set of linear segments. It would therefore be reasonable to use it instead of the current function.

3.8 Alternate Primitives

This section gives a brief outline of several alternate primitives. None of these alternates has been adequately studied so only guesses can be made on how useful they would be for the programmer.

3.8.1 Alternate Combining Functions

With a multiplicative combining function one activity could linearly modulate another. Such modulation can be used to implement *controlled contrast-enhancement* groups (see section 7.1.4), and to inhibit and excite large subnetworks. Since communication is the bottleneck on the current implementation on the Connection Machine, the addition of a multiplicative combining function would not have a large effect on the running time. On the more efficient hardware suggested in section 8.6, the addition of multiplication would have a greater effect - it is not yet clear what the magnitude of the effect would be.

A pair of combining functions that might be well suited for "reasoning with uncertainty" are $B+$ and $B*$. These operators are defined as follows:

```
(I1 B+ I2) = (1 - (1 - I1)(1 - I2))
(I1 B* I2) = (I1 * I2)
```

Where I1, I2 and the results are values between 0 and 1. These operators cost approximately the same as a multiplication. It is hard to say without more experimentation how useful they are.

3.8.2 Alternate Spreading Functions

Rather than spreading a copy of the output value, it might be reasonable to implement a function that either decays as the fan-out gets bigger, or depends on the activity at the other end of the link. Such a scheme has been suggested in [Reggia85]. In Reggia's model, the activity flowing out of a node is constant so that as the fan-out gets larger, each element gets a smaller amount of the activity. He also suggests that the amount of activity which each output gets should be proportional to the activity at the other end of the link.

3.8.3 Bidirectional Links

Neural networks and connectionist models have used bidirectional links in two ways. The first is in a strict manner which forces all the links to have equal weights in both directions. With such a constraint, one can prove some important properties about the stability of the networks [Hopfield82]. The second allows for the links to have different weights in both directions and is used to get a mutually excitatory effect [Feldman84, Reggia85].

In the first case it would be desirable to have bidirectional links as primitives. In the second case a higher level construct **make-bidirectional-link** can be created out of the primitive construct **make-link**. Little efficiency is gained by including a primitive **make-bidirectional-link**.

Bidirectional links are often harder to program than unidirectional links because it is hard to account for all the feedback effects.

3.8.4 Links That Learn

Many researchers have noted that static networks can learn by modifying the weights on links between the nodes [Rosenblatt58, Minsky78, Feldman82, Hinton85, Rumelhart85].

In these models there are two ways in which a link can decide to change its weight parameters: purely from local information - the history of the activity of its neighbors [Hinton85, Rumelhart86], or from some global stimulus - this stimulus usually signifies punishment or reward [Rosenblatt58].

It would not be hard to extend the AFL-1 primitives to allow for modifications of the weights, and for most methods it would be cheap to implement. This thesis does not include links that learn since allowing the networks to learn would make the initial study of a programming language too complex.

3.8.5 Capacitance (State)

The nodes in AFL-1 have no internal state (Capacitance). Once a node sends the **ovalue** out of its output it has lost that value: the value calculation on the next clock will not depend on it. In the applications AFL-1 is designed for, this is not a problem because it is easy to simulate capacitance by creating a link from a node back to itself. If capacitance is used in a major part of the network then it would be worthwhile including capacitance in the primitives and allowing the user to specify the value of the capacitance when defining the node.

Chapter 4

Node Groups

A *node group* (NG) defines an abstract collection of primitives. To create network structure (nodes and links) from a node group, the node group is instantiated. The relation between node groups and node group instances is analogous to the relation between object types (flavors, classes) and object instances in object oriented languages such as Smalltalk [Goldberg83]. Unlike most object oriented languages, the definition of NGs can include arguments and control code so the structure of each instance of a single NG can differ substantially. The node groups therefore should not be thought of as fixed static structures, although their instances should be. Node groups are instantiated at compile time. At run time, the **network-processor** only sees the primitives.

Sections 4.1 through 4.3 discuss how to define and instantiate node groups, how to reference elements in different instances, and how to create tangled hierarchies. Section 4.5 discusses some generally useful groups that are predefined in the AFL-1 environment, and section 4.6 discusses some libraries of more task specific groups.

By hierarchically defining NGs in terms of other NGs, one can build large AFNs with a modest amount of code.

4.1 Defining Node Groups

The **defgroup** form is used to define node groups. The syntax is the same as the **defun** function of Common Lisp [Steele84].

```
(defgroup group-name argument-list &rest body)
```

The argument-list can include the symbols &key, &optional and &rest with the same semantics as in Common Lisp.

To instantiate a group, one appends the group name to "**make-**" and calls this derived function along with an instance name and the NGs arguments.

The following example shows how node groups are defined and instantiated.

```
(defgroup mutually-inhibit (first-node second-node)
  (make-inhibitory-link first-node second-node)
  (make-inhibitory-link second-node first-node))
```

```
(defgroup position (object)
  (cond ((ocean-bound object)
         (make-node 'at-sea)
         (make-node 'at-dock)
         (make-mutually-inhibit 'dock-sea (get-node 'at-dock)
                                          (get-node 'at-sea)))
        ((person object)
         (make-node 'at-home)
         (make-node 'at-work)
         (make-mutually-inhibit 'home-work (get-node 'at-home)
                                           (get-node 'at-work)))))

(defgroup speed ()
  (make-node 'fast)
  (make-node 'stopped)
  (make-mutually-inhibit 'fast-stopped (get-node 'fast)
                                       (get-node 'stopped)))

(defgroup ship ()
  (make-position 'ship-position 'ocean-bound)
  (make-speed 'ship-speed)
  (make-mutually-inhibit 'dock-fast
                         (get-node '(ship-position at-dock))
                         (get-node '(ship-speed fast))))

(make-ship QE-II)
(make-ship SS-FRANCE)
```

Although not particularly useful, this examples shows many interesting points about node groups (NGs). Firstly, it shows how NGs can be nested. In the example, the **ship** NG includes an instance of the **position** NG which includes an instance of the **mutually-inhibit** NG - the hierarchy is show in figure 4.1. Secondly, it shows that nodes can be created conditionally. Any Common Lisp code is allowed within a **defgroup** and can be used to direct the creation of nodes. The instances of the **position** NG will differ depending on whether the instantiator is **sea-bound** or a **person**.

Thirdly, the example shows the use of *smart data* (data-procedure integration). Each instance of the ship node group is a smart data structure - if at run time the **(QE-II ship-speed fast)** node gets turned on somehow, the **(QE-II ship-position at-dock)** will get turned off due to the **mutually-inhibit** links between these two nodes. Mutual exclusion is discussed further in chapter 7.

Fourthly, the example shows how *open modules* are used. Although instantiating NGs creates modular structure, there are no restrictions on which instances can make

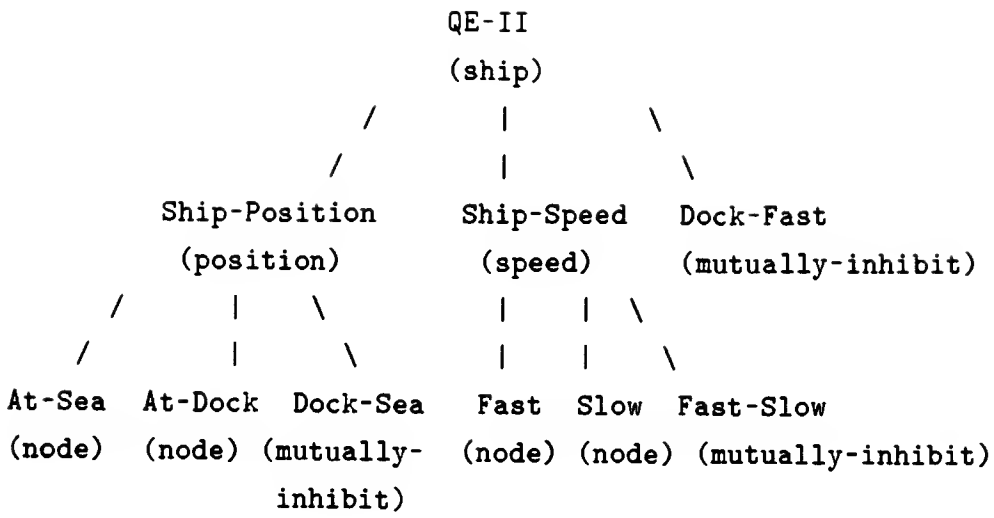


Figure 4.1: The Node Group Hierarchy of the Ship Example.

links to which other instances. In the example, the instances of the **ship** NG makes links to both **position** and **speed** instances. This thesis call such modules *open modules.* One might claim that since random links are allowed, debugging will be hard, but with appropriate debugging tools this is usually not the case.

In AFL-1 no nodes or links are created until a "**make-**" command is called at top level. In the example, the two **make-ship** commands will create all the necessary nodes and links.

4.2 Referencing Nodes

For an NG instance to make a link to, or from, a node in another NG instance, it must have a pointer to that node. There are basically two ways that an instance can get such a pointer. The first is that the pointer can be passed in as an argument. In the example, the **mutually-inhibit** NG instances get their pointers this way - **first-node** and **second-node** are pointers to two nodes. The second way is to use the **get-node** function. The **get-node** function takes as an argument a list of names (symbols) that leads along the hierarchy to the node of concern, and **get-node** returns a pointer to that node.

```
(GET-NODE node-name-list)
```

Get-node works as follows. The **node-name-list** must either be a symbol or a list of symbols. If it is a symbol, then **get-node** will look for the node in the instance in which it is called. If the **node-name-list** is a list, then **get-node** will go down the list as it goes down levels of the hierarchy. In the ship example, the **(get-node (ship-speed fast))** call in the definition of the **ship** NG will search down through the **ship-speed** instance for a **fast** instance. If the **node-name-list** begins with "<" symbols, then **get-node** will go up a level of the hierarchy for every "<" it encounters. Using this scheme, **get-node** can reference any node in a tree from any other node in a tree.

When passing node references as arguments, it is usually best to pass pointers rather than names since names refer to different nodes depending on where **get-node** is applied.

For brevity, the two functions **make-inhibitive-link-relative** and **make-excitatory-link-relative** are defined in AFL-1. The function

```
(MAKE-EXCITATORY-LINK-RELATIVE from-name to-name)
```

is equivalent to

```
(MAKE-EXCITATORY-LINK (get-node from-name) (get-node to-name))
```

The abbreviations **make-ir-link** and **make-er-link** are also defined.

It is sometimes desirable to reference a node from the root of the hierarchy rather than from the current position. The function

```
(GET-NODE-ABSOLUTE node-name)
```

exists for this purpose. Absolute referencing is rarely used inside node group definitions but is often used by the programmer to study the network.

To reference a NG instance rather than an actual node **get-vertex** is used.

It is sometime desirable to have a list of all the nodes of a specific form. For example, in the ship world we might want a list of the pointers to all the **(* ship-speed fast)** nodes, where * will match any symbol. The **get-node-list** function is used for this purpose.

```
(GET-NODE-LIST name-matching-list)
```

The **get-template-node-list** function is used when only a fixed set of elements need to be matched. An example of its use is:

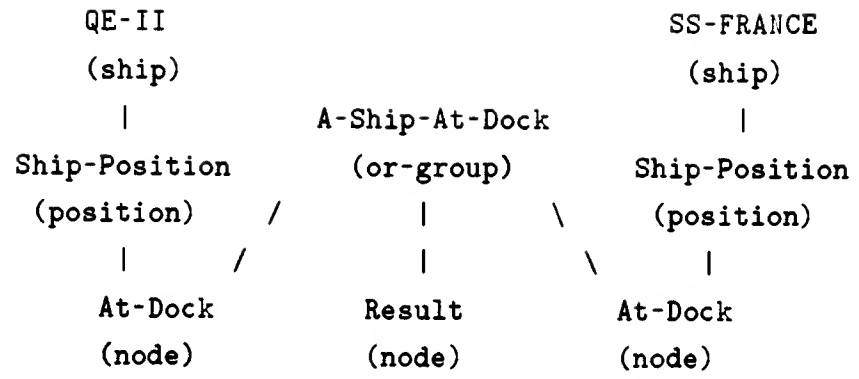


Figure 4.2: Example of a Tangled Hierarchy: **A-Ship-At-Dock**.

```
(GET-TEMPLATE-NODE-LIST '(*wild* ship-speed fast)
                        '(QE-II SS-FRANCE QUEEN-MARY))
```

This will return a list of pointers to three **fast** nodes.

4.3 Tangled Hierarchies

It is often useful to define a function over a collection (group) of already existing nodes. These existing nodes might have been instantiated by several different node groups. For example, in the ship world, it might be necessary to find if there are any ships at dock. To do this, we require a node that becomes active when any of the nodes **(* ship-position at-dock)** are true (* matches any ship in existence). The required node would take an "or" of all the **at-dock** nodes. To do this an **or** groups is defined; a diagram of part of the desired hierarchy including the **or** group is shown in figure 4.2.

To differentiate between nodes that will be created by instantiating a node group, and nodes that can be referenced directly but that are not created by an instance - as with the **(* ship-position at-dock)** nodes from the **or** group instance - AFL-1 separates *member* and *internal* nodes of a group. In the NGs defined in the previous sections, all nodes were internal nodes. When an **or** group is instantiated over the ships at dock, all the **(* ship-position at-dock)** nodes are member nodes of the new group.

By having member nodes as well as internal nodes AFL-1 allows multiple hierarchies (tangled hierarchies). Such tangled hierarchies add programming power and make studying and debugging Activity Flow Networks easier since they allow the user to

view a set of member nodes from several contexts. The (**QE-II ship-position at-dock**) node could be studied either from the **QE-II** context or from the **a-ship-at-dock** context. The following two functions are used to include member nodes in the definition of a node group.

```
(ADD-MEMBER name node-pointer)
(ADD-MEMBERS name list-of-node-pointers)
```

The **add-member** function adds a single member node and gives it the name specified by the name argument. The **add-members** function creates a subgroup that has the name specified in the argument, and places the elements given by **list-of-node-pointers** in that subgroup. The elements are given the names **member-1, member-2, ... , member-n**. To define the **mutually-inhibit** group given in section 4.1 with member nodes, the **add-member** function could be used as follows.

```
(defgroup mutually-inhibit (first-member second-member)
  (add-member 'first-member first-member)
  (add-member 'second-member second-member)
  (make-inhibitive-link first-member second-member)
  (make-inhibitive-link second-member first-member))
```

To define the **or** group, the **add-members** function would be used since the number of members can differ on each instantiation.

```
(defgroup or-group (member-list)
  (add-members 'input member-list)
  ... )
```

With these definitions, one references the first member of the **dock-fast** instance of the **mutually-inhibit** NG with **(QE-II dock-fast first-member)** and references the first member of the **a-ship-at-dock** instance of the **or** group with **(a-ship-at-dock input member-1)**.

4.4 For Variables

For-variables is a macro used to loop over sets of values that a variable can have. For most purposes, the **for-variables** macro will expand as follows.

```
(for-variables ((x '(dog cat mouse))
                (y '(north south east north)))
  (....))

(dolist (x '(dog cat mouse))
  (dolist (y '(north south east north))
    (....))
```

Since there are no variables in AFL-1, this form is used quite often.

4.5 Predefined Groups

Several node groups are included in the AFL-1 environment. This section outlines some of these NGs. The description of these groups introduces the types of groups that might be useful.

4.5.1 The Fan-In Groups (Or, And, Max, Min and Sum)

Groups that take information from many nodes and supply it to a single node are called *fan-in* groups. All the fan-in groups have an internal **result** node which receives the information. The function:

```
(MAKE-OR-GROUP instance-name members
               &optional (threshold *active*))
```

creates a group with the member nodes specified by the members argument, and an internal node **"result"**, which is activated when any of the member nodes are activated. For example, the call

```
(MAKE-OR-GROUP 'a-ship-at-dock
               (GET-NODE-LIST '(* ship-position at-dock)))
```

will create a node **(a-ship-at-dock result)**, which will be active when one of the nodes **(* ship-position at-dock)** is active. The **or-group** considers active to be the activity-element *active* (8/8th), but by using the optional **threshold** argument, the required activity can be changed to any element of the activity set. **Max-nodes** are used to implement **or-groups**.

The other fan-in groups are similar to the **or-group** and have the following properties. The **and-group** will activate its result node if all of its member nodes have an

activity of at least the group's threshold. The **sum-or-group** uses **sum-nodes** instead of **max-nodes** so that no single member of the group has to have an activity as high as the group's threshold, but the sum of the activities must be. The **sum-and-group** is like the **sum-or-group** but the threshold is linearly dependent on the number of members. The **max-group, min-group** and **sum-group** simply set the result node to the max, min and sum of their members.

4.5.2 The Fan-Out Groups (Activate and Inhibit)

Groups that take information from a single node and send it to many nodes are called *fan-out* groups. All the fan-out groups have an internal **source** node which the information is taken from. The function:

```
(MAKE-ACTIVATE-GROUP instance-name members &optional
                     (source-spec '(*active* *nil*
                                    *active* *active*)))
```

creates a group with the member nodes specified by the members argument, and an internal node **"source"**, which if activated will send activity to all of the member nodes of the group. For example, the call

```
(MAKE-ACTIVATE-GROUP 'put-all-ships-at-dock
                     (GET-NODE-LIST '(* ship-position at-dock)))
```

will create a node **(put-all-ships-at-dock source)**, which if activated will activate the nodes **(* ship-position at-dock)**. The optional argument **source-spec** can be used to change the activity sent to the member nodes as a function of the activity sent to the **source** node. **Max-nodes** are used for the **source**.

The **inhibit-group** works in the same way except that it creates inhibitory instead of excitatory links to the member nodes.

4.5.3 Mutual Interaction Groups (Mutual Inhibition and Excitation)

While the fan-in and fan-out groups are concerned with bringing information to or spreading information from a single point, each member of a *mutual-interaction* group interacts symmetrically with all the other members. This means that between every pair of nodes in a group, there is at least one path in each direction along which information can flow.

The most important of the mutual interaction groups are the mutually exclusive groups (ME-groups). In ME-groups, if one member is activated the other members are inhibited. There are two types of ME-groups currently available in AFL-1, a winner-take-all group (WTA-group) and a contrast-enhancement group; Chapter 7 discusses the function of these groups.

```
(MAKE-WTA-GROUP instance-name members
                &optional (hysteresis-threshold *nil*))
(MAKE-CONTRAST-ENHANCEMENT-GROUP instance-name members
                                 contrast-factor)
```

There are three other mutual-interaction groups available in AFL-1, the **mutually-excitatory** group, the **lower-bound** group and the **upper-bound** group. In a **mutually-excitatory** group, the members excite each other. In a **lower-bound** group, the average activity of all its members is kept above some lower bound. In a **upper-bound** group, the average activity of all its members is kept below some upper bound.

```
(MAKE-MUTUALLY-EXCITATORY-GROUP instance-name members
                                excitation-factor)
(MAKE-LOWER-BOUND-GROUP instance-name members
                        lower-bound)
(MAKE-UPPER-BOUND-GROUP instance-name members
                        upper-bound)
```

4.6 Libraries

As in conventional languages, node groups relevant to particular classes of applications can be collected into libraries. This section suggests some libraries that might be useful. These libraries are only suggestions and are not meant in any way as an analysis of whether the various functions can actually be implemented adequately on the static networks created by AFL-1 nor as an adequate description of the systems discussed. References are given to descriptions of the systems.

4.6.1 Semantic Networks

Several researchers have shown how semantic memories can be implemented on architectures similar to Activity Flow Networks [Collins75, Fahlman79, Hinton81b, Shastri85].

The most complete of these is in [Shastri85]; the following node groups might be used to create semantic networks of the sort he suggests.

```
(MAKE-CONCEPT concept-name)
(MAKE-PROPERTY property-name possible-values)
(MAKE-IS-A-LINK link-name parent-concept-name child-concept-name
                weight)
(MAKE-PROPERTY-LINK link-name concept-name property-name value
                    value-weight concept-weight)
```

In this system **concepts** correspond to things such as **dog, cat,** and **person**, **properties** correspond to things such as **has-hair-color** and **eats**, and the **values** a **property** has might be **red, brown** or **orange**.

4.6.2 Natural Language

Connectionist implementations of natural language interpreters are discussed in [Selman85, Waltz85, Cottrell85]. AFL-1 cannot implement the system suggested in [Waltz85] since his system builds networks on the fly (dynamically). The systems suggested in [Selman85] and [Cottrell85] use static precompiled networks and can therefore be created using node groups. The following node groups might be used to create the grammar rules for the syntax parser suggested in [Cottrell85].

```
(MAKE-TOKEN-TYPE token-type-name)
(MAKE-CONSTITUENT constituent-name)
(MAKE-ROLE role-name possible-constituent-list)
(MAKE-GRAMMAR-RULE rule-name constituent-name role-list)
```

A set of grammar rules written with these groups might look as follows.

```
(MAKE-TOKEN-TYPE 'VERB)
(MAKE-TOKEN-TYPE 'NOUN)
(MAKE-TOKEN-TYPE 'PRONOUN)

(MAKE-CONSTITUENT 'sentence)
(MAKE-CONSTITUENT 'verb-phrase)
(MAKE-CONSTITUENT 'noun-phrase)

(MAKE-ROLE 'subject '(noun-phrase))
(MAKE-ROLE 'predicate '(verb-phrase))
(MAKE-ROLE 'main-verb '(VERB))
```

```
(MAKE-ROLE 'direct-object '(noun-phrase))
(MAKE-ROLE 'indirect-object '(noun-phrase))
(MAKE-ROLE 'head '(NOUN PRONOUN))

(MAKE-GRAMMAR-RULE 'simple-sentence 'sentence
                   '(predicate))
(MAKE-GRAMMAR-RULE 'subject-sentence 'sentence
                   '(subject predicate))

(MAKE-GRAMMAR-RULE 'simple-verb-phrase 'verb-phrase
                   '(main-verb))
(MAKE-GRAMMAR-RULE 'complex-verb-phrase 'verb-phrase
                   '(main-verb indirect-object direct-object))

(MAKE-GRAMMAR-RULE 'simple-noun-phrase 'noun-phrase
                   '(head))
```

A library of natural language groups could also include groups to deal with lexical and semantic constraints.

4.6.3 Production System

Chapter 5 discusses a production system implemented in AFL-1. The groups defined in that chapter, along with several other rule groups, can be collected into a library.

Some of the groups of this library might be:

```
(MAKE-SIMPLE-PARAMETER parameter-name)
(MAKE-PARAMETER parameter-name values)
(MAKE-ANTECEDENT-RULE rule-name if-parts then-parts)
(MAKE-CONSEQUENT-RULE rule-name if-parts then-parts)
(MAKE-META-RULE rule-name if-parts then-parts)
```

4.6.4 Vision

There are several aspects of vision that can be abstracted into groups of nodes. Since there is still debate on what the primitives of vision are, it is not obvious what the "base" groups should be. A vision library could include groups based on several approaches. Vision NGs based on [Hinton85] might include:

```
(MAKE-INPUT-RETINA-PLANE instance-name height width)
(MAKE-FEATURE feature-name description)
(MAKE-RETINOTOPIC-PLANE instance-name input-plane
```

```
                        feature-list height width)
(MAKE-OBJECT-PLANE instance-name feature-list height width)
(MAKE-MAPPING-PLANE instance-name retinotopic-plane object-plane
                    feature-list)
(MAKE-LETTER letter-name description)
(MAKE-LETTER-MAP instance-name object-plane letter-list)
```

4.7 Conclusion

To make programming large systems practical, a language must allow the user to divide the system into smaller parts. In connectionist networks the most natural way to make the division is into collections of nodes. In previous work with connectionist models, such collections have been given several different names and have been used in several different ways. These collections include layers [Rumelhart85], frames [Feldman85], planes [Hinton85], spaces and cliques [Touretzky85], groups [Selman85], levels and buffers [Cottrell85], and polynemes, xenomes and isonomes [Minsky86]. In some cases they are used to separate different sections of the network so the network is easier to understand (frames [Feldman85] and planes [Hinton85]), and in other cases they are used so that the structure can be repeated within the network (buffers [Cottrell85] and groups [Selman85]). The definitions of these collections have always been very task specific.

The node group abstraction of AFL-1 tries to unify these previous methods. With a single method it will be easier to communicate code, build on other peoples work, join different systems, and formalize results.

Although the node groups defined in this chapter work to some extent there is room for considerably more work. There are a few problems with NGs that should be mentioned. Using the hierarchy created by NGs it is sometimes hard to express the relations among objects. For example, if **John, the-chair** and **is-on** are three vertices in the NG hierarchy, it is not obvious where to put nodes that specify relations among the objects. Do nodes that signify "**John is-on the-chair**" get put below **John, is-on** or **the-chair**? Another related difficulty is with deciding on the order of the hierarchy: what goes above what. For example, if we have **bathrooms** and **kitchens** and they can be small or large, which of the hierarchies shown in figure 4.3 should one use?

```
(MAKE-OBJECT-PLANE instance-name feature-list [ width])
(MAKE-MAPPING-PLANE instance-name retinotopic-plane object-plane
                    feature-list)
(MAKE-LETTER letter-name description)
(MAKE-LETTER-MAP instance-name object-plane letter-list)
```

4.7 Conclusion

To make programming large systems practical, a language must allow the user to divide the system into smaller parts. In connectionist networks the most natural way to make the division is into collections of nodes. In previous work with connectionist models, such collections have been given several different names and have been used in several different ways. These collections include schemas [Rumelhart86], frames [Fahlman85], planes [Hinton85], spaces and clipdes [Touretzky85], groups Selman85], levels and buffers [Cottrell85], and polynemes, xenomes and isonomes [Minsky80]. In some cases they are used to separate different sections of the network so the network is easier to understand (frames [Feldman85] and planes [Hinton85]), and in other cases they are used so that the structure can be repeated within the network (buffers [Cottrell85] and groups [Selman85]). The definitions of these collections have always been very task specific.

The node group abstraction of APE tries to unify these previous methods. With a single method it will be easier to communicate code, build on other peoples work, join different systems, and formalize results.

Although the node groups described in this chapter work to some extent there is room for considerably more work. There are a few problems with NGs that should be mentioned. Using the hierarchy created by NGs it is sometimes hard to express the relations among objects. For example, in the NG hierarchy, it is not obvious where to put nodes that specify relations among the objects. Do nodes that signify "John is-on the-chair" get put below John, is-on or the-chair? Another related difficulty is with deciding on the order of the hierarchy: what goes above what. For example, if we have bathrooms and kitchens and they can be small or large, which of the hierarchies shown in figure 4.3 should one use"

Chapter 5

CIS - A Production System

This chapter describes the Concurrent Inference System (CIS), a production system implemented in AFL-1. The description brings out some issues in the design of activity flow networks and gives an example of how to program with the AFL-1 language. Production Systems are well suited for an example since their techniques are non-trivial but relatively well understood.

In the system described in this chapter, and in the other examples described in this thesis, the reader should pay particular attention to the following issues:

- The use of node groups to create repeated structures in the network.
- The use of active structures.
- The constraints imposed by the networks being static.
- The way one can make an AFN focus - this is a problem that plagues all concurrent systems.
- The explicit representation of information and control flow.

The inference mechanism used by CIS includes most of the capabilities of both the Mycin system [Davis77] and of OPS5[Forgy81]. A currently implemented version is capable of forward and backward chaining, which happen concurrently; using meta-rules of the sort described by Davis [1980]; and reasoning with "uncertainty", a variation of Zadeh's rules [1965]. Although not implemented, suggestions on how the system can select the most pertinent questions to ask according to specified heuristics, and on how to implement explanation facilities are given. With 100,000 rules on the current implementation of CIS, a global inference step takes less than two seconds. A global inference step is the time needed for a single change to percolate through all the rules.

Some features of existing production systems are hard or expensive to implement with AFNs. These include the use of high-precision numbers, the ability to dynamically bind arbitrary values to a parameter, general purpose unification algorithms, and the ability to dynamically create an arbitrary number of instances of an object. It will be argued in this chapter that many practical production systems do not require

these features. The rule sets of Mycin [Davis77], R1 [McDermott80] and Prospector [Gaschnig80] can be implemented without them, with no loss of power. On the other hand, highly numeric applications that require 64 bit floating point numbers would be impractical to implement with AFNs.

CIS will be described incrementally by building up through the following features to the final system.

- Forward Chaining - a simple system that activates propositions in the **then-parts** or rules with active **if-parts**.
- Input and Output - nodes are added that allow the system to communicate to the outside world.
- Backward Chaining - a mechanism that allows the system to search from a goal proposition for propositions that assert that goal.
- Parameter/Value Pairs - the ability to represent assertions in terms of parameter/value pairs instead of atomic propositions.
- Meta-Rules - rules that control the invocation of other rules.
- Objects - the ability to define objects and then create instances of those objects - assertions are now object/parameter/value triplets.
- Fuzzy Reasoning - assertions can have analog values which signify to what extent they are true; methods for combining these values are included.

Each of these section shows the necessary AFL-1 code. Sections 5.10 and 5.11 discuss focus and explanation but do not give any code. Section 5.12 discusses issues of activity flow programming and the Concurrent Inference System.

Information on the syntax and semantics of AFL-1 can be found in chapter 3 and 4.

5.1 Antecedent Reasoning (Forward Chaining)

Two types of objects exist in the antecedent reasoning system: parameters and rules. The parameters are atomic propositions - for example, **has-hair, is-ungulate** or **color-red**; rules are used to define relations among these parameters. The parameter and rule objects are defined using the **defgroup** form discussed in chapter 4. Figure 5.2 shows the AFL-1 definitions of the node groups. Each rule added to the

activity flow network (AFN) will be an instance of the **rule** NG and each parameter will be an instance of the **simple-parameter** NG. In this purely antecedent system, the NGs only have a single node and the network resembles a truth maintenance system [Doyle79].

Using these NGs, the following parameter and rule instantiations will get compiled into the network shown in figure 5.1.

```
(make-simple-parameter 'has-hair)
(make-simple-parameter 'gives-milk)
(make-simple-parameter 'is-mammal)
(make-simple-parameter 'has-hoofs)
(make-simple-parameter 'is-ungulate)
(make-simple-parameter 'has-black-stripes)
(make-simple-parameter 'is-zebra)

(make-antecedent-rule 'animal-rule-1
  '(if has-hair)
  '(then is-mammal))

(make-antecedent-rule 'animal-rule-2
  '(if gives-milk)
  '(then is-mammal))

(make-antecedent-rule 'animal-rule-3
  '(if is-mammal has-hoofs)
  '(then 'is-ungulate))

(make-antecedent-rule 'animal-rule-4
  '(if is-ungulate has-black-stripes)
  '(then is-zebra))
```

The **active** nodes of the rule groups act as **and** gates (MIN), and the **asserted** nodes of the parameter groups act as **or** gates (MAX).

5.2 Communication With The Outside World

The only way an AFN can communicate to the outside world is through the **input** and **output** nodes (see section 3.4). The following description assumes that, at runtime, the **host** computer serves as an interface between the AFN and the user by noticing when an **output** node is active and by activating **input** nodes. In CIS the **host** computer will only be used to do things that are necessarily serial such as communication with the user. If the system can get a set of inputs in parallel, as with some sensor based

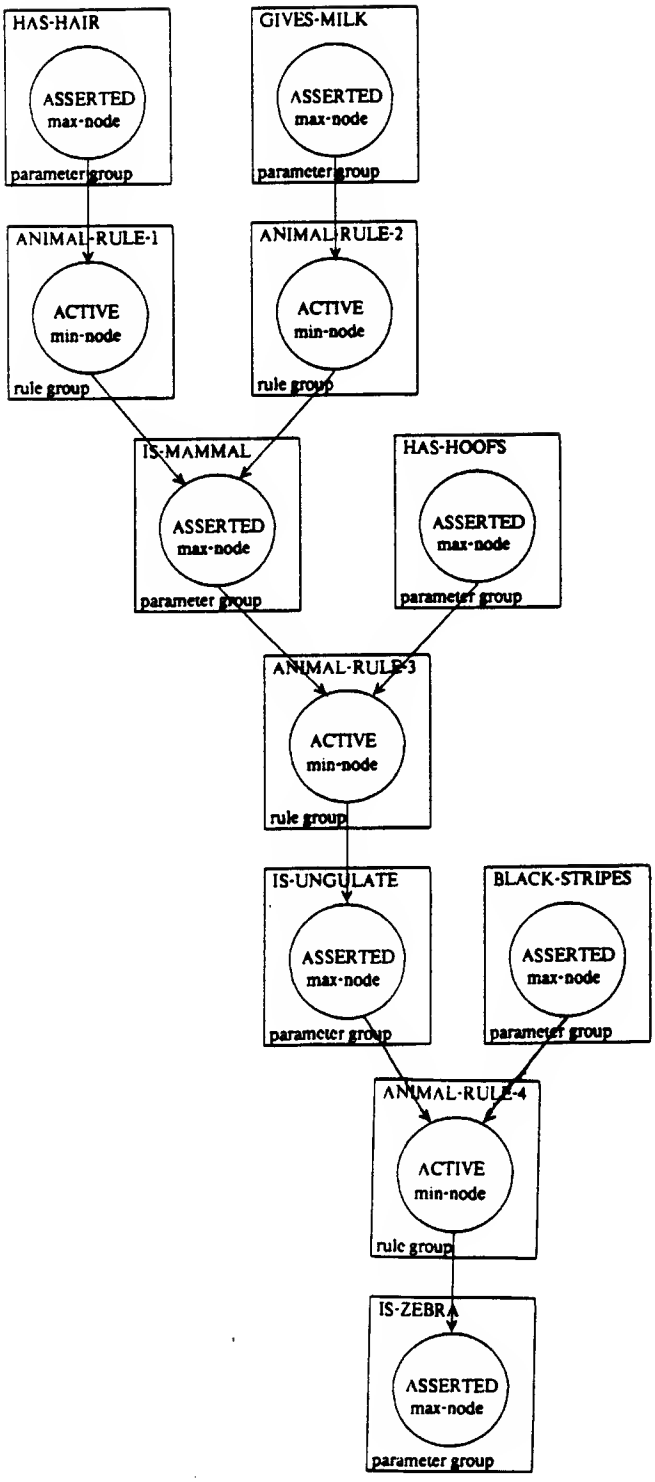


Figure 5.1: The Network Generated by Four Animal Rules.

reminder of syntax

```
(make-node name &optional (Threshold *active*) (Slope *nil*)
                          (Rise *active*) (Saturation *saturation*))
(make-er-link from-node-name to-node-name &optional (Weight *active*))
(defgroup group-name argument-list &rest body)
```

the node groups

```
(defgroup simple-parameter ()
  (make-max-node 'asserted))

(defgroup antecedent-rule (if-parameters then-parameters)
  (make-min-node 'active)
  ;; make a link from each if parameter to the active node
  (dolist (parameter if-parameters)
     (make-er-link `(< ,parameter asserted) 'active))
  ;; make a link from the active node to each of the then parameters
  (dolist (parameter then-parameters)
     (make-er-link 'active `(< ,parameter asserted))))
```

Figure 5.2: The Definition of the Simple-Parameter and Antecedent-Rule NGs.

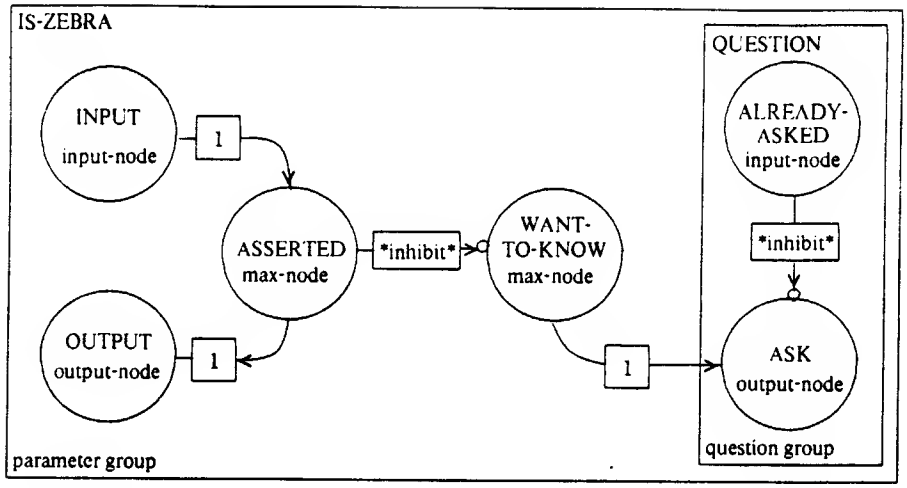


Figure 5.3: The Zebra Instance of the Simple-Parameter NG Including I/O and Question Asking.

production systems such as PDS [Fox83], these inputs are attached directly to the **input** nodes of the AFN. Such direct connections will not be discussed in this thesis.

To provide CIS with I/O, we add two nodes to the **simple-parameter** NG, an **input** and an **output**. Using the **input** nodes the **host** computer can assert parameters, and using the **output** nodes it can check whether parameters are asserted.

5.3 Asking Questions

In many applications one wants a production system to ask questions rather than having it rely on spoon fed information. To allow CIS to ask questions, we add a node, the **want-to-know** node, and a node group instance, the **question-group**, to the **simple-parameter** NG. The **question-group** is supplied by the AFL-1 language and has two nodes, the **ask** node and the **already-asked** node. Figure 5.4 shows the AFL-1 definition of the new **simple-parameter** NG, and figure 5.3 shows the network created by instantiating it.

The activation of the **want-to-know** node signifies that for some reason (given in the next section), the system wants to know if the parameter is true. The **want-to-know** node will activate the **ask** node unless the **already-asked** node is active.

So that the **host** computer knows at run time where the **ask** nodes are, the **question-group** supplies it, at compile time, with a data structure for each parameter. This data structure includes a question string, and pointers to the **ask**, **already-asked** and **input** nodes of that parameter. At run time the **host** computer executes the fol-

```
(defgroup question-group (input-node question-string)
  (make-latched-input-node 'already-asked)
  (make-output-node 'ask)
  (make-ir-link 'already-asked 'ask *saturation*)
  (add-to-host-record (get-node 'ask) (get-node 'already-asked)
                      input-node question-string))

(defgroup simple-parameter ()
  (make-max-node 'asserted)
    ;; the input
  (make-input-node 'input)
  (make-er-link 'input 'is-true)
    ;; the output
  (make-output-node 'output)
  (make-er-link 'is-true 'output)
    ;; the want-to-know node
  (make-node 'want-to-know)
  (make-ir-link 'asserted 'want-to-know *saturation*)
    ;; instantiating the question group
  (make-question-group 'question (get-node 'input)
                       *this-instance-name*)
  (make-er-link 'want-to-know '(question ask)))
```

Figure 5.4: The Definition of the Simple-Parameter NG Including I/O and Question Asking.

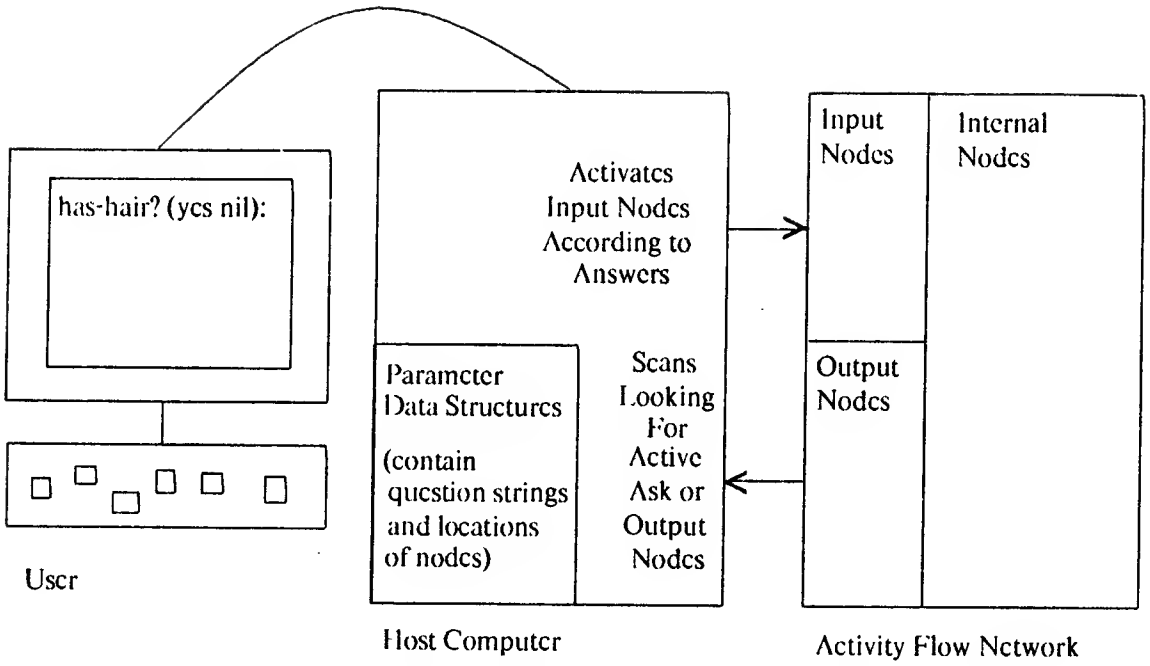


Figure 5.5: The Interface Between the AFN and the User.

lowing routine. A diagram of this interaction is shown in figure 5.5.

```
do forever
  for all parameters P
    if ask-node of P active
      print out question string of P
      wait for answer from user
      activate the input node of P according to answer
      activate the already-asked node of P
```

When a parameter is asserted by the **host** computer at run time, it takes some time for the effect of the change to propagate through the network. The unit of time required for a changed assertion to propagate is called a **global inference step**. This unit consists of the length in links of the longest path the assertion travels times the time taken by **afl-step** (see section 3.5). In CIS the longest path length is twice the depth of the rules, and since rules are rarely 10 deep, a **global inference step** can usually consist of 20 **afl-steps**. To guarantee that the user is not asked unnecessary questions, CIS takes a **a global inference step** between each question.

After each **global inference step** the **host** computer picks at random a parameter with its **ask** node active to ask. Section 5.10 discusses how the AFN instead of the **host** can decide which question gets asked.

5.4 Consequent Reasoning (Backward Chaining)

Often in Production Systems one is trying to find out whether certain "goal" parameters are true or not, and it is desirable for the system to only ask questions that could lead to the assertion of one of these goal parameters. This sort of reasoning is often called backward chaining, goal-directed reasoning or consequent reasoning. The consequent reasoning used in CIS differs slightly from conventional consequent reasoning since it happens concurrently with the antecedent reasoning.

To include goal directed reasoning in CIS, we add a **want-to-know** node to the **rule** group, analogous to the one in the **parameter** group, and we add some extra links between the two types of groups. Figure 5.7 shows the AFL-1 definition of the new **rule** group and figure 5.6 shows the network generated by one of the animal rules. The extra links point in the opposite direction from the previous links: the *want-to-know* activity flows in the opposite direction from the *true* activity.

The backward chaining done by CIS has two advantages over many current production systems. Firstly, since the "want-to-know" and "true" activity flow across different links, the antecedent and consequent reasoning happen concurrently. Consider a medical diagnosis program which is searching for disease X, and meanwhile stumbles across all the symptoms for a perhaps much more serious disease Y. Most current systems would ignore this, and perhaps not come to disease Y for a long time. CIS would find Y while it starts searching for X.

Secondly, unlike Prolog and Mycin, the inferences are not restricted by recursion to follow the same path as the control. The backward chaining done by CIS is therefore easier to extend. Three examples of such extension are:

- Backward chaining links can easily be excluded from some rules; this allows a mix of antecedent and consequent rules.

- With very few changes meta-rules of the sort discussed by Davis [80] can be implemented; such rules are discussed in section 5.6.

- It is easy to make "fuzzy" backward chaining links and use these links as one of many heuristics that guide the search rather than force it. This will be discussed in section 5.10.

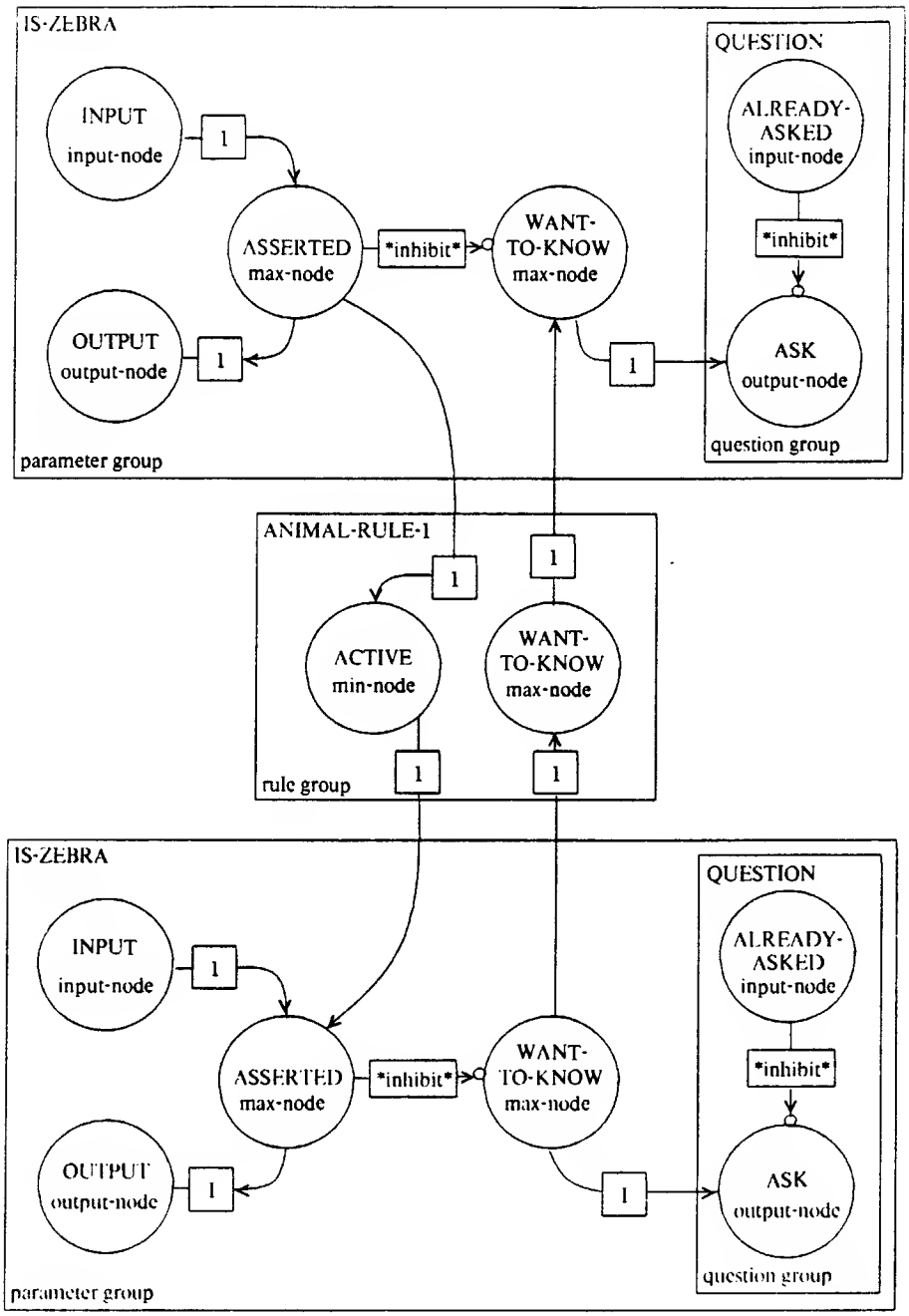


Figure 5.6: The Network Generated by A Consequent Animal Rule.

```
(defgroup consequent-rule (if-parameters then-parameters)
  (make-min-node 'active)
  (make-node 'want-to-know)

  ;; make the forward and backward links to the if parameters
  (dolist (parameter if-parameters)
    (make-er-link `(< ,parameter asserted) 'active)
    (make-er-link 'want-to-know `(< ,parameter want-to-know)))

  ;; make the forward and backward links to the then parameters
  (dolist (parameter then-parameters)
    (make-er-link 'active `(< ,parameter asserted) weight)
    (make-er-link `(< ,parameter) 'want-to-know)))
```

Figure 5.7: The Definition of the Consequent-Rule Node Group.

5.5 Parameter Value Pairs

With the **asserted** node of the **simple-parameter** group, the only values a parameter can have are true or nil. If one desires to distinguish between false and unknown, or among several values, more nodes must be added.

CIS includes a separate node for each value a parameter can have. This imposes the following two restrictions on how values can be used in CIS:

- The values a parameter can be bound to must be specified at compile time. At run time the user selects one of the specified values.
- The set of values can't be large - in particular, infinite. A set of 100 or so values is reasonable but 32-bit integers are not.

In most rule sets the first restriction is not a problem, and in many current production systems including Mycin, KEE and Prospector one usually defines at compile time the values a parameter can be bound to since this helps prevent errors. For the values of parameters that differ every time a rule set is run, such as the patient's name in Mycin, the **host** computer can bind the name to a generic value, such as **name-1**, and make inferences using the generic value. The system might include a few different generic values so it could make inferences about a few different patients at the same time.

The second restriction precludes the use of integers and floating point numbers. Although most other production systems allow integers and floating point numbers,

many applications don't need them. For example Mycin only uses integer values for age, body temperature and dates of last examination or immunization. For these parameters 32 bit integers are not needed. A 100 or so values are plenty for these parameters and qualitative measures such as **child, young-adult, adult, or senior-citizen** for age, and **low, normal, high or critical** for body temperature will probably be sufficient. Similarly, the rule sets of Prospector and R1 do not require high-precision numbers. If an application requires 64 bit floating-point values, the problem is not well suited for AFNs.

To include **values** in CIS we define the **value** group. A **parameter** group creates an instance of this **value** group for each of its values. Figure 5.9 shows the AFL-1 definition of the **value** group and the new definition of the **parameter** group. The animal rules given in section 5.1 are reformulated in terms of parameter/values below. Figure 5.8 shows the network generated by **covering** instance of the **parameter** NG.

```
(make-parameter 'skin-pattern '(black-stripes dark-spots tawny-color))
(make-parameter 'covering '(feathers hair))
(make-parameter 'animal-class '(mammal bird reptile))
(make-parameter 'eating-class '(ungulate carnivore))
(make-parameter 'animal-name '(cheetah zebra ostrich))
(make-parameter 'ped-type '(claws hoofs))

(make-consequent-rule 'animal-rule-1
  '(if (covering hair))
  '(then (animal-class mammal)))

(make-consequent-rule 'animal-rule-2
  '(if (animal-class mammal) (ped-type hoofs))
  '(then (eating-class ungulate)))

(make-consequent-rule 'animal-rule-3
  '(if (eating-class ungulate) (skin-pattern black-stripes))
  '(then (animal-name zebra)))
```

By default, the **parameter** NG places a mutually exclusive group (see chapter 7) around all its values, so only one can become active at a time. Using the keyword argument (:exclusive NIL), this group can be left out.

This new formulation of the animal rules has two advantages over the previous formulation. Firstly, it will reduce the number of questions because it will ask a generalized form. CIS will ask the question "What is the skin-pattern of the animal?", rather than "Does the animal have black-stripes?". Secondly, because of the mutually exclusive group it knows that if a parameter has one value, it does not have the others.

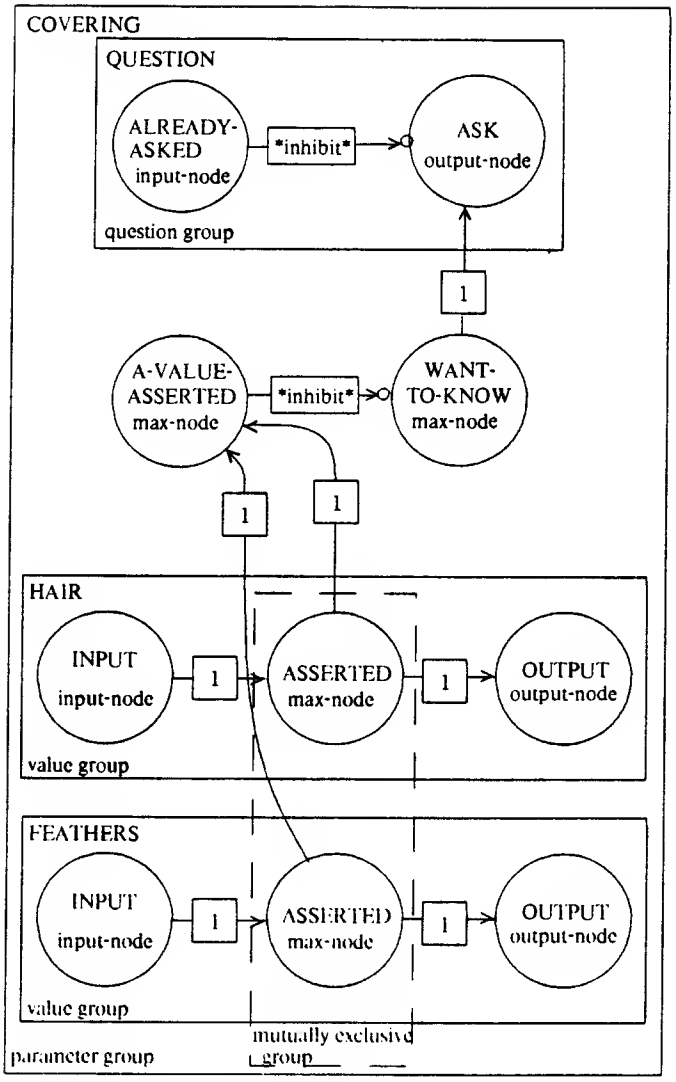


Figure 5.8: The Covering Instance of the Parameter NG.

```
(defgroup value ()
  (make-node 'asserted)
  (make-latched-input-node 'input)
  (make-output-node 'output)

  (make-er-link 'input 'asserted)
  (make-er-link 'asserted 'output))

(defgroup parameter (values &key (exclusive t))
    ;; creates a value group for each value
  (dolist (value values) (make-value value))
    ;; is activated when any of the values is active
  (make-or-group 'a-value-asserted (get-node-list '(* asserted)))
    ;; makes sure that only one of the values is active
  (if exclusive
    (make-me-group 'value-selection (get-node-list '(* asserted))))
    ;; the want-to-know node
  (make-node 'want-to-know)
    ;; deactivates the WANT-TO-KNOW node if one of the values is known
  (make-ir-link '(a-value-asserted result) 'want-to-know *saturation*)
    ;; creates the question group with all values as possible answers
  (make-question-group 'question (get-node-list '(* input)))
    ;; activates the ASK node if the WANT-TO-KNOW node is active
  (make-er-link 'want-to-know '(question ask)))
```

Figure 5.9: The Definition of the Value and New-Parameter Node Groups.

```
(defgroup meta-rule (if-parameters then-parameters)
  (make-min-node 'active)

  ;; make links from if parameters.
  (dolist (parameter if-parameters)
    (make-er-link '(< ,@parameter) 'active))

  ;; make links to the then parameters
  ;; make the link inhibitive if "not" is specified.
  (dolist (parameter then-parameters)
    (if (equal (first parameter) 'not)
        (make-ir-link 'active '(< ,@(cdr parameter)))
        (make-er-link 'active '(< ,@parameter)))))
```

Figure 5.10: The Meta-Rule Node Group.

For the animal rules the system knows that if an animal is a mammal, it is neither a bird nor a reptile; this saves the user from having to add these rules.

5.6 Meta Rules - Rule Set Activation

In practice, it is important to have task specific rules that control the invocation of other rules [Davis79], [Gaschnig82]. An example of such a rule is: *"if the patient has stepped on a rusted nail, then ask questions about tetanus* (activate the tetanus rules)." Davis [80] named such rules, "meta-rules", and Prospector [Gaschnig82] names them "contextual relations".

To include this type of rule in CIS we add the **meta-rule** NG which only differs from the **antecedent-rule** NG in that it does not append "asserted" onto the if and then parts. Figure 5.10 shows the AFL-1 definition of the **meta-rule** NG. This change allows a rule to specify the **want-to-know** node of a parameter as its **then** part. By controlling the **want-to-know** node of a parameter one can control the activation of a set of rules. For example, the tetanus rule given above could be implemented by having a rule with "stepped-on rusted-nail asserted" as its if part and "tetanus want-to-know" as its then part. Figure 5.11 shows the network created by such a rule. This network causes all the questions relevant to tetanus to be asked if the parameter "stepped on rusted nail" becomes asserted.

Meta-rules can also be used to deactivate or order the activation of rule sets. Examples of such meta-rules are:

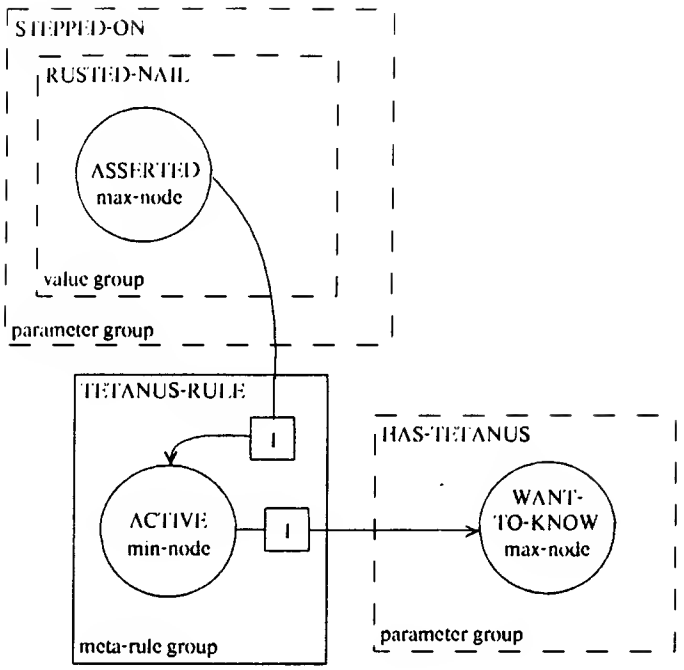


Figure 5.11: The Network Created by the Tetanus Meta-Rule. In the diagram the dashed groups signify that not all the nodes in those groups are shown.

```
(make-rule beach-bum-rule-1
  (if (near-beach false))
  (then (surf-is-good (not want-to-know))))

(make-rule beach-bum-rule-2
  (if (beach-is wicked-good))
  (ordered-then (how-to-get-there want-to-know)
                (do-you-want-to-come want-to-know)))
```

The first of these capabilities is included in CIS. The second is not but is not difficult to add.

This section discussed domain specific methods for controlling the line of reasoning by manipulating the **want-to-know** nodes, section 5.10 discusses some domain independent methods.

5.7 Objects And Instances

When a rule set includes several instances of an object that all obey the same properties (rules), it is convenient to create a single set of rules which are valid for all instances. To allow for this, systems such as Mycin, OPS5 and KEE have generic objects (often part of the object, attribute, value triplet) which are used in the rules. The system can

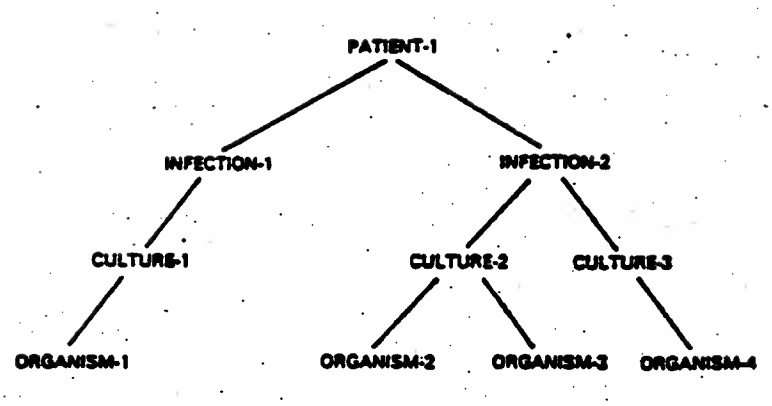


Figure 5.12: The Object Hierarchy of Mycin [Davis77].

create multiple instances of these objects. For example, in Mycin, a culture is defined as an object and during a session several instances of this object are created. These instances obey the same rules but can have different values bound to their parameters and can coexist without interacting. In Mycin, the objects are grouped in a hierarchy; this hierarchy is shown in figure 5.12.

CIS allows multiple instances of an object by creating multiple networks for that object. This requires than one knows at compile time the maximum number of instances that will be created. In most practical applications, this is not a problem since the number of instances created does not vary greatly from one use of a system to the next. For example, in Mycin, we know that there is only going to be a single patient and the patient will usually have at most three cultures taken each with possibly three organisms.

Similarly, in R1 there is only a finite number of backplanes, each with a finite set of slots which only accept one of a finite set of modules. The combination of these finite sets may seem large, but when analyzed, is not unreasonable. Assuming it takes 100 links to specify a module and a maximum of 100 modules need to be placed, then 10,000 links are needed to represent the modules. If there is a maximum of 100 slots for the modules, another 10,000 links are needed, one for each slot/module combination. Since all other sets of links will require no more than the same order of magnitude in number, on the order of 100,000 links are needed for the whole system. As chapter 8 discusses, networks of a million links are reasonable for machines that exist today, and networks of a trillion links will probably be practical in ten years from now. The 100,000 links is therefore not pressing the current limit.

Prospector also uses a limited form of object instantiation. "potential matches are relatively few in number and can be precomputed ... our current implementation tacitly assumes that any variable can be bound in only one way, so that, for example,

only one entity composed of galena would be allowed"[Duda78]. This limited form of instantiation, and even the less limited form where only a fixed number of entities composed of galena would be allowed, can be implemented in CIS.

Even with a fixed number of instances compiled into the static network a few things can be done if additional instances are required at run time. Firstly, an extra instance sub-network could be added. This might take some time but would not have to be done often. Secondly, one of the instance sub-networks could be arbitrated between two or more instances - this would cause them to run sequentially.

To define objects in CIS, one defines a node group (NG) for that object. To create instances of the object, one instantiates that NG. Such a node group will henceforth be called an **object** NG and includes in its body all the parameters that the object has and all the rules that are particular to the object. At run time, the **host** computer assigns particular names of new instances to particular instance sub-networks.

An **animal object** NG can be defined for the animal rules given in section 5.1. This would allow the system to reason about several animals at the same time. The **animal object** NG would be defined and instantiated as follows:

```
(defgroup animal ()
  (make-parameter 'skin-pattern '(black-stripes dark-spots tawny-color))
  (make-parameter 'skin-cover '(feathers hair))
  (make-parameter 'animal-class '(mammal bird reptile))
  (make-parameter 'eating-class '(ungulate carnivore))
  (make-parameter 'animal-name '(cheetah zebra ostrich))
  (make-parameter 'ped-type '(claws hoofs))

  (make-rule 'animal-rule-1
    '(if (skin-cover hair))
    '(then (animal-class mammal)))

  (make-rule 'animal-rule-2
    '(if (animal-class mammal) (ped-type hoofs))
    '(then (eating-class ungulate)))

  (make-rule 'animal-rule-3
    '(if (eating-class ungulate) (skin-pattern black-stripes))
    '(then (animal-name 'zebra))))

(make-animal 'first-animal)
(make-animal 'second-animal)
(make-animal 'third-animal)
```

Object NGs as well as allowing multiple instances to be created, allow a clean way to separate sets of rules into modules.

5.8 Variables

Although rules defined in an **object** NG (henceforth object rules) allow references within a single instance of an object, they can't make references to different instances or objects. Object rules allow rules with a single variable such as:

```
(if (x animal-class mammal) (x ped-type hoofs))
(then (x eating-class ungulate))
```

but do not allow rules with multiple variable such as:

```
(for-variables ((x in animal-instances)
                (y in animal-instances))
  (if (x is-zebra) (x father y))
  (then (y is-zebra)))

(for-variables ((x in animal-instances))
  (if (equal-number-of (x is-zebra) 3))
  (then (pack-of zebras)))

(for-variables ((x in animal-instances)
                (y in people-instances)
                (z in object-instances))
  (if (y on x) (x is-horse) (y holding z) (z is-long-stick))
  (then (y playing-polo)))
```

This is because the rules within the definition of an **object** NG are instantiated once each time an instance of that object is created and only refer to the parameters of already existing instances. The later three rules given above require that a rule of an instance has a link to a different instance that might be created at a later time.

To create rules that make references across instance boundaries, the same **consequent-rule** NG can be used but it is used outside of the definition of an **object** group and the **for-variables** form (defined in section 4.4) is used to reference the instances. All the rules that are used to make references among instances of a particular object are grouped together in a **class** NG. So for example, to add a **father** rule to the animal rules the following code could be used. Figure 5.13 shows some of the network this code creates.

```
(defgroup animal-class (animal-instances)
  (for-variables ((x animal-instances))
    (make-animal x))

  (for-variables ((x animal-instances)
                  (y animal-instances))
```

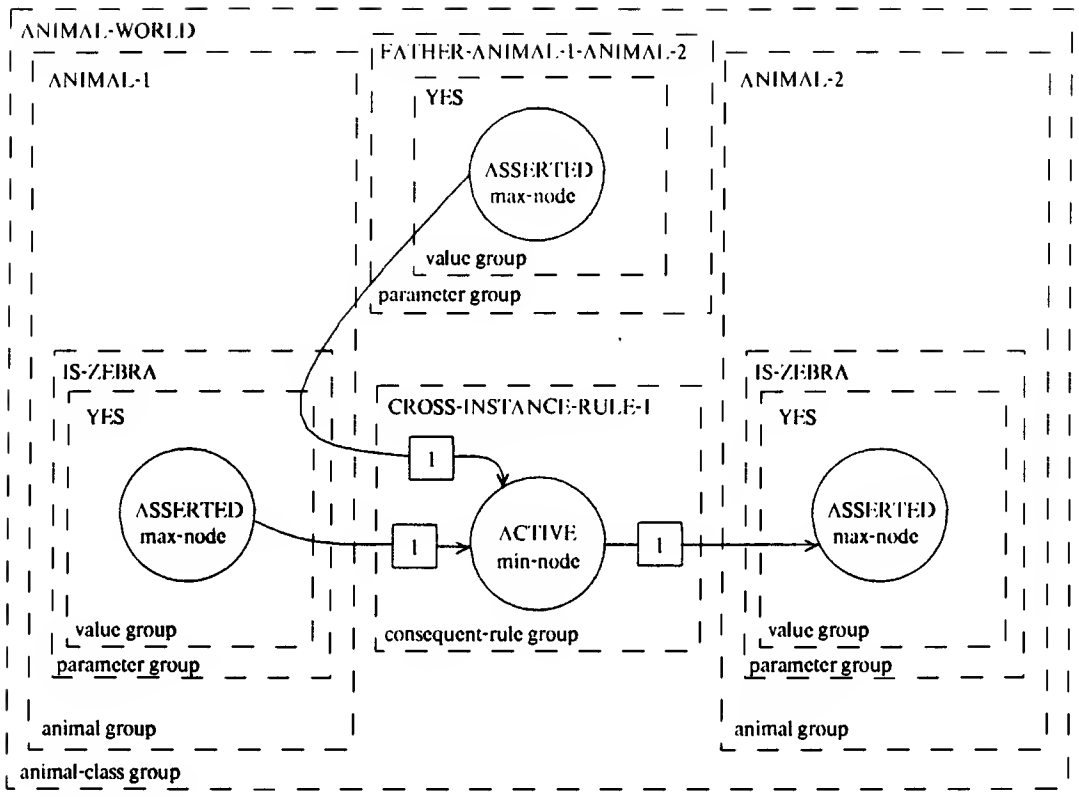


Figure 5.13: Example of Cross Instance Rule.

```
(make-parameter (combine 'father x y) '(yes no))

(make-consequent-rule 'cross-instance-rule-1
                      '(if (,(combine 'father x y) yes)
                           (,x is-zebra yes))
                      '(then (,y is-zebra yes)))))

(make-animal-object 'animals '(animal-1 animal-2))
```

To create rules that cross **class** boundaries, the rules are defined at yet a higher level. An example of a cross-class rule is:

```
(for-variables ((x people-instances)
                (y animal-instances))
  (make-parameter (combine 'on x y) '(yes no))
  (make-consequent-rule 'cross-class-rule-1
                        '(if (,(combine 'on x y) yes)
                             (,y is-horse yes))
                        '(then (,x is-horseback-riding yes))))
```

Figure 5.14 shows the hierarchy of node groups discussed so far.

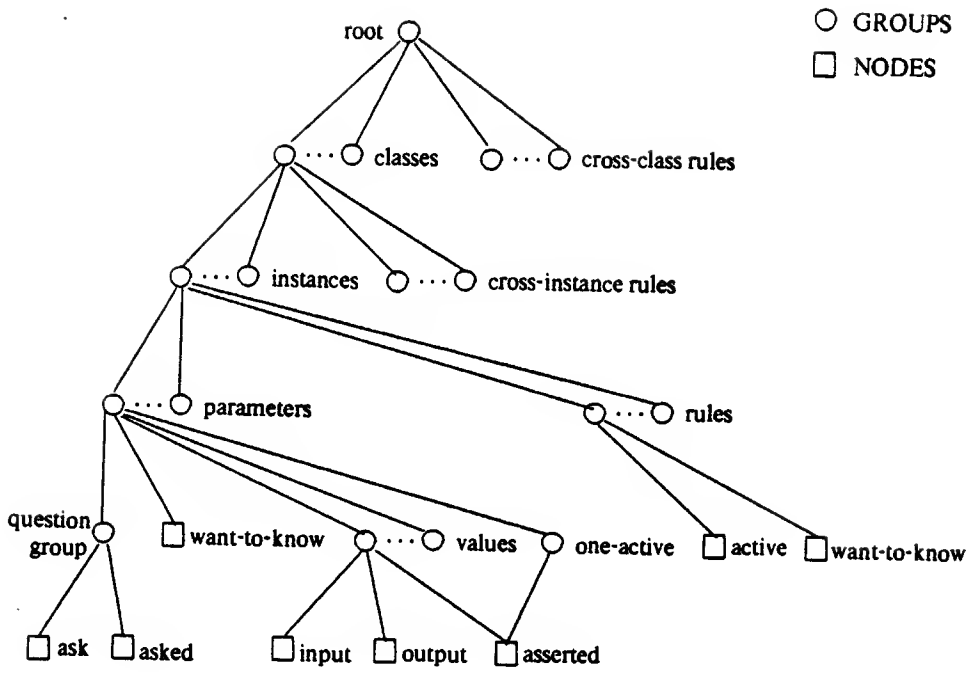


Figure 5.14: The Hierarchy of Node Groups in CIS.

The cross-instance and cross-class rules create an instance of the **parameter** and **rule** groups contained within their form for each combination of their variables. This means that many **rule** and **parameter** instances can be created if there are several variables each with several bindings. The user should take care when creating cross-instance and cross-object rules.

5.9 Inexact Reasoning

Inexact reasoning is the ability to reason about assertions that are not known with absolute certainty. So far CIS has only allowed values of 1 or 0 on its nodes - each node had to be completely on (active) or completely off (inactive). To include inexact reasoning in CIS we allow the assertions to have intermediate values.

There have been several proposals concerning how to implement the combining functions for rules of inexact or uncertain assertions. The most popular of these are variations of Zadeh's rules of *fuzzy sets* [65], and variations on the methods of *Bayesian* decision theory. The first type interprets the values associated with a parameter loosely as "certainty factors", "degrees of membership" or "degrees of belief"; the second type interprets the values more precisely as probabilities. The combination functions of fuzzy sets are relatively easy to implement on an activity flow network since they are simple

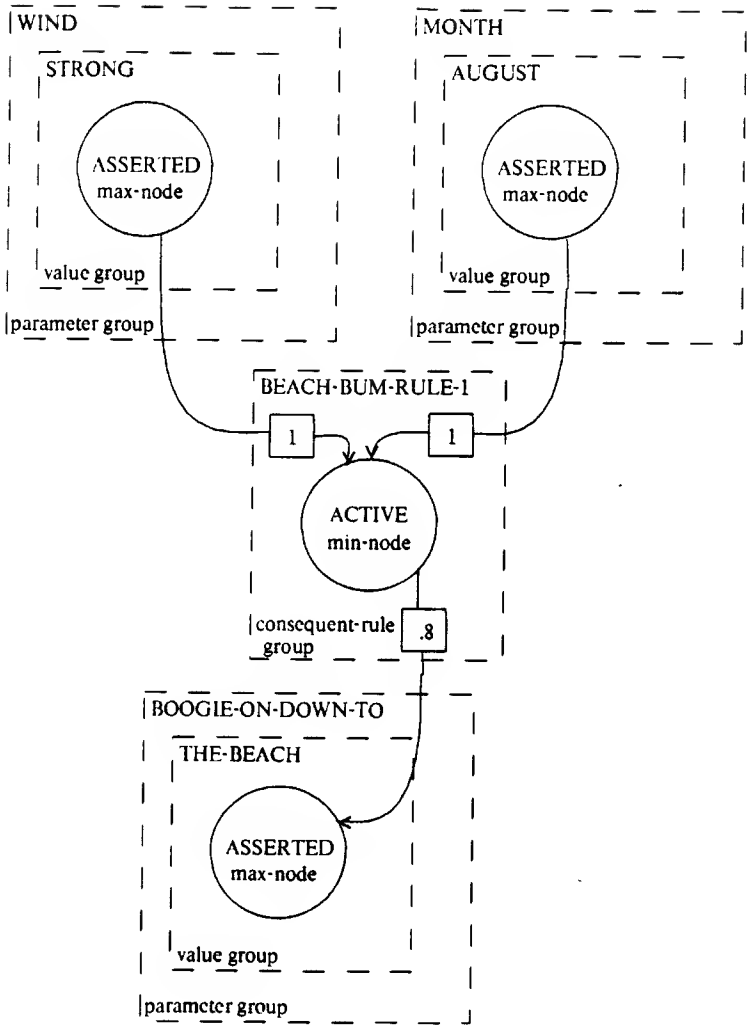


Figure 5.15: Inexact Beach-Bum Rule.

(Min and Max) while those of Bayesian decision theory are expensive to implement unless the functions are built into the primitives.

CIS uses a variation of Zadeh's rules similar to that used by Mycin's rules and Prospector's "logical relations". The **minimum** of the input activities is used for conjunction (AND), and the **maximum** is used for disjunction (OR). If the result is below some threshold, then no activity is passed on. The **active** node of a **rule** group thus takes the Minimum (AND) of the activities of all the elements of its if-part, and the **asserted** node of a **value** group takes the Maximum (OR) of all its inputs.

Rules now require a weight of belief for each of the **then** parameters. This weight is placed on the link between the **active** node of the **rule** group and the **asserted** node of the **value** group. Figure 5.16 shows the AFL-1 definition of the new **value** and **rule** NGs. Figure 5.15 shows the network that the following rule compiles into.

```
(defgroup value ()
  (make-node 'asserted .2 1 .2)
  (make-latched-input-node 'input)
  (make-output-node 'output)

  (make-er-link 'input 'asserted)
  (make-er-link 'asserted 'output))

(defgroup consequent-rule (if-parameters then-parameters)
  (make-min-node 'active .2 1 .2)
  (make-node 'want-to-know)

  ;; make the forward and backward links to the if parameters
  (dolist (parameter if-parameters)
    (make-er-link `(< ,@parameter asserted) 'active)
    (make-er-link  'want-to-know `(< ,(first parameter) want-to-know)))

  ;; Make the forward and backward links to the then parameters.
  ;; The forward link has a weight specified by the third element of
  ;; the parameter.
  (dolist (parameter then-parameters)
    (make-er-link 'active `(< ,@(cdr parameter) asserted)
                          (first parameter))
    (make-er-link `(< ,(second parameter) want-to-know) 'want-to-know)))
```

Figure 5.16: The Value and Consequent-Rule NGs for Inexact Reasoning.

```
(rule beach-bum-rule-1
  (if (wind strong) (month august))
  (then (.8 boogie-on-down-to the-beach)))
```

An alternate OR combining function is:

```
Or  : result = MIN ( SUM( sources ) , 1 )
```

This combining function has the advantage that the more active sources there are the more active the result will be. It might be reasonable to use this OR function and to use MIN for the AND function. Currently rule groups with either combining function are available; **make-sum-consequent-rule** creates rules with a SUM combining function while **make-consequent-rule** creates rules with a MAX combining function.

Everything up to this point in the chapter has been implemented and works. What is discussed in the following two sections has not yet been implemented.

5.10 Focus

This section discusses mechanisms that select the most pertinent questions to ask the user first. Such selection concerns asking related questions together and asking more critical questions first, and will henceforth be called *focusing*. Like the Meta-Rules discussed in section 5.6, the focusing methods control the invocation of rules, but instead of using task specific rules to invoke other rules, they use task independent mechanisms.

On serial production systems that backtrack, a limited form of focusing comes at no cost. When serially backtracking through the inference tree, related questions are asked together by virtue of the local tree walk. It has been noted [Duda78, Miller82] that this sort of focusing is of a restrictive type and does not allow many of the powerful focusing methods humans use. Again, a general method of conventional programming, recursion, allows for a restricted form of a desirable effect but without significant changes, can't be extended. Some systems such as Prospector [Gaschnig82], Internist [Miller84] and AM [Lenat78] include more advanced focusing ideas based on heuristic methods. These systems rate the best, or most *interesting* parameter by using a set of heuristic rules. Each rule adds to the interest value of parameters according to some heuristic, and the parameter with the greatest interest value is selected.

Such techniques can be added to CIS easily, and because they run concurrently, will not slow the system down. An *interest measure* can be added to each parameter in CIS by allowing the **want-to-know** node to have any activity level instead of just being on or off. So that the AFN can select the parameter that wants to be known the most, a mutually exclusive group (see chapter 7) is placed around the **ask** nodes of all the parameters. With the ME-group, the **host** computer will only have a single choice of which question to ask. The remainder of this section discusses various focusing heuristics which could be used to activate the **want-to-know** nodes.

5.10.1 The More the Better

The first heuristic is to more highly activate the **want-to-know** nodes of parameters that lead to more results. For example, in the animal rules given is section 5.1, consider the goal "find if the animal is either a zebra or a giraffe"; the parameter **is-ungulate** leads to both **is-zebra** and **is-giraffe.** If parameters with more prospects are activated more, this parameter (is-ungulate) will be asked first, and if answered in the negative, will be the only question ever asked. The system can save asking many questions by using this trivial heuristic - recursive backtracking systems don't use it.

This heuristic method comes for free in CIS. Since **want-to-know** nodes are **sum** nodes, they add the values on the **want-to-know** nodes that have links to them.

5.10.2 Related Set Activation

The following method can be used to group related questions together. A **related-set** group is defined that takes as its arguments a list of related parameters, and puts a **mutually-excitatory** groups around the **ask** nodes of these parameters. To prevent the network from uniformly saturating, a **mutually-inhibitory** group (also called mutually-exclusive group) is created around the sets of related parameters so that no more than one of the sets is active at a time. The mutually-excitatory and mutually-inhibitory groups will activate all the questions of one set before the system goes on to the next set. The height of hysteresis (discussed in chapter 7) of the **mutually-inhibitory** groups, could be tuned to best decide when it is worth switching focus between two related sets.

5.10.3 Strict Ordering

The focusing mechanism used by many backtracking serial production systems can be added as another heuristic to CIS. By including this heuristic we only help guide the reasoning rather than force it a particular way. To include such a mechanism, nodes would be added to the **rule** node groups that order the activation of the **want-to-know** nodes of the rule's **if** parameters. A path would also be necessary from the antecedents back to the rule to state that the particular antecedent has been exhaustively searched. In such a scheme three types of activity would flow between the parameters and rules - want-to-know, truth, and searched. The first flows backwards the other two flow forward.

5.10.4 Close to Answer

Another heuristic is to prefer rules that have most of their antecedents satisfied. This heuristic can be implemented by having an additional node in each **rule** NG that detects the difference between the current truth activation coming into the rule and the activation needed to fire the rule. Rules that are closer to being fired will send more want-to-know activity to their antecedents than rules that are far from being fired.

5.11 Explanation

It is important for a production system to be able to explain what it is doing and justify its decisions in a way that can be understood by the user. Such a capability is both important when developing a rule set so that it is easy to debug, and when using a rule set so that the user feels more confident about the decisions and can catch mistakes. This section outlines a way to add some simple explanation facilities to CIS.

In two places during the execution of CIS it makes sense to let the user ask about the reasoning process. One is when the system asks a question and the user is interested in why the question is being asked. The other is when the system changes an assertion and the user is interested in why the assertion has changed.

The **host** computer can allow the user to ask "Why" whenever an assertion changes or a question is being asked. The ability to answer these questions can be added to the AFN by creating nodes and links in the **rule, parameter** and **value** groups that allow a trace of the causes of the state in question. In the case of tracing assertions, the desired links would go backward through the rules and if a question is asked by the user, activate a node in each parameter that supports the given assertion. In the case of tracing questions, the desired links would go forward through the rules and, if a question is asked by the user, activate a node in each parameter that led to the activation of the given **want-to-know** node. Each parameter would have an output node which is activated when that parameter has an effect on the changed assertion or question asked. The system could execute this search one level of rules at a time.

More flexible techniques of the sort discussed in [Swartout81] could be implemented without great effort.

5.12 Discussion

There are several issues of activity flow processing with respect to the Concurrent Inference System (CIS) that are worthy of review or discussion.

5.12.1 Limitations

Many abstractions that are natural in symbolic processing languages and which most programmers take for granted, are unnatural and expensive to implement with activity flow networks. Such abstractions include recursion and general purpose variable binding. Likewise, many abstractions that are natural in activity flow languages are unnatural and expensive to implement with Symbolic Processing Languages. There

are several consequences of these expenses on the design of CIS. These consequences are summarized here.

Restrictions imposed on CIS by AFNs:

- High precision numbers and operators to manipulate them are not supplied.
- The possible values of the parameters must be given at compile time.
- The maximum number of instances of an object must be given at compile time.
- Rules with a large number of variables each which is quantified over a large range of values are expensive.

Features included in CIS because of its implementation on AFNs.

- Forward and backward reasoning are run together. Advantages of this are discussed in section 5.4.
- Complete AND/OR searches are executed between questions.
- Large sets of heuristic rules are used to select which question to ask.

5.12.2 Extensions

Many of the limitations of CIS can be avoided by expanding the model. For example, to manipulate high-precision numbers, one could use a model that intermixes data flow and activity flow networks, and to dynamically create instances of an object, one could use a system that creates extra network structure as it is needed. Another interesting extension would be a method that changes the weights on the forward links between the rules and parameters according to some measure of how well the network made its inferences.

5.12.3 The Maximum Numbers of Rules

By making some assumptions about the rules and parameters and imposing a limit on the time the user is willing to wait between questions, an upper bound on the number of rules can be given. With the following assumptions it is possible to include 100,000 rules in CIS.

- The maximum time a user is willing to wait is 2 seconds.
- The maximum depth of inferences in the system is 20 rules.

- The average rule has three antecedent and two consequent parts.
- The average parameter has five values.
- There are five times as many rules as parameters.

This is an order of magnitude larger than any existing expert system.

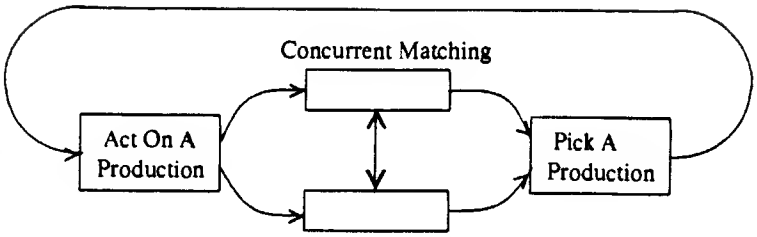
For 100,000 rules, the above assumptions require a network of 2 million links. With 2 million links an **afl-step** requires .05 seconds which allows 40 of them to be executed within a **global-inference-step** (two seconds). The running time of an **afl-step** for other numbers of links can be found in section 8.3.

5.12.4 Concurrency

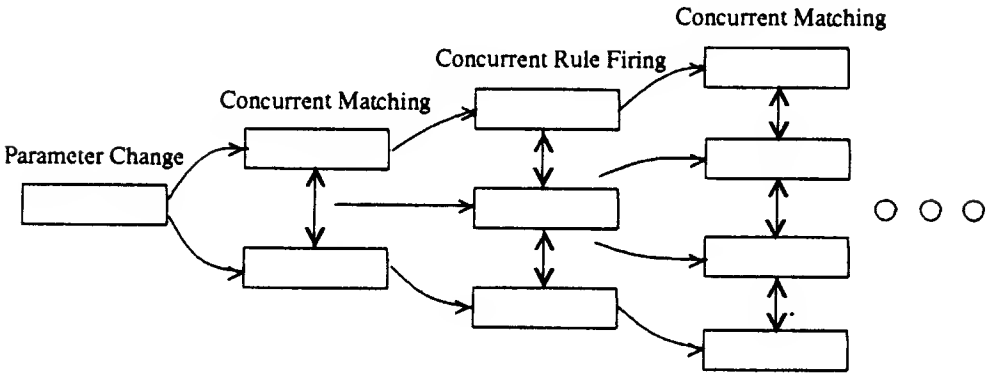
Researchers have argued that the speedup one can achieve by implementing rule sets on concurrent rather than serial systems is at best a constant factor [Forgy84, Oflazer84]. They make these arguments based on Production Systems developed for single processor machines, in particular OPS [Forgy78], and only consider a very limited interpretation of a production system. In particular, they consider a model that forces the selection of rules through a single channel so only one rule can fire at a time (the "conflict resolution" stage). The concurrency available in this model is limited to matching the antecedent parts of productions to changes made to "working memory" by a single production. Once these matches are made, and a "conflict set" is selected, only a single production is chosen and fired. This sort of concurrency is pictured in figure 5.17a.

CIS takes advantage of many more sorts of concurrency. Among the sorts of concurrency CIS uses are:

- Subrule and subparameter concurrency - within the rules and parameters all the parts act concurrently. For example, at the same time that a value activates the rules it is connected to, it deactivates all the other values of its parameter, activates the **parameter-known** node, and activates its output node.
- Concurrent matching - all the antecedent parts of a rule are matched concurrently. In fact it only takes a single **afl-step** to match every rule in the system. This is possible because the variable references are compiled out so the variable slots do not become bottlenecks.
- Concurrent forward propagation - all the rules can propagate their inferences concurrently. There can potentially be a large fan-out so that a single change



a) Type discussed by [Oflazer84] and [Forgy84].



b) Type used by CIS.

Figure 5.17: Two Types of Forward Chaining Concurrency.

could propagate to make thousands of changes in just a few **afl-steps**. Note that this sort of concurrency is much richer than the sort considered by [Forgy84]; figure 5.17 shows the difference between the two types.

- Concurrent backward propagation (Searching) - a completely concurrent AND/OR search is done from the "goal" parameter to the parameters that can effect it. Unlike in the concurrent implementations of logical inference systems discussed in [Douglass85] and [Murakami84], there is no problem with concurrent "AND" searching. This is again because the variables are compiled out.

- Forward and Backward propagation happen together - as mentioned in section 5.4, this has some significant advantages.

- Concurrent question selection - For the system suggested in section 5.10, the system does a concurrent search of a single question to ask the user.

Although CIS allows all these sorts of concurrency, how well the system takes advantage of these depends on the particular rule set being used. Since no significant rule

sets have been implemented in CIS, no data is available on the effect of rule sets on how efficiently the concurrency of the AFN is used.

The main reason OPS allows only a single rule to fire at a time is because some rule sets programmed in the language are action oriented - rules signify actions rather than inferences. With action oriented rule sets it is often incorrect to execute certain rules simultaneously. For example in R1 it would be incorrect to put both **board-1** and **board-2** into **backplane-slot-3** if backplanes can only accept a single board.

The conflict among action oriented rules is not a valid motivation for always restricting the system to fire rules serially, for two reasons. Firstly, only subsets of all the action oriented rules conflict and one should only serialize those subsets. Secondly, one should separate action and inference oriented rules and only consider serializing the first type. Conflict sets might be serialized in CIS by defining a node group that places a **mutually-exclusive** group around the **active** nodes of conflicting rules. Such a group could be instantiated for each set of conflicting rules (actions).

5.12.5 Data Procedure Integration

In CIS all the rules and parameters are active modules. Instead of being manipulated by an inference engine as in most production systems, they contain their inference mechanism in their structure. This has both speed and programming advantages over production systems that separate the inference engine and working memory. In CIS the inference engine does not serve as a bottleneck because much of the work is done directly at the data. It has programming advantages for the same reasons as object oriented languages do. Since the functionality lies with objects rather than in the middle of some large piece of code it is easier to see what a particular rule does. It is also easier to make special case rules that differ in some way from the other rules. This entails defining a new object (node group) rather than playing with guts of the inference engine.

5.12.6 Meta-Control

Although a AFN cannot manipulate or study its own structure, it can control the activation of nodes or areas of the structure. In general, this can consist of activating or inhibiting whole areas of the network or ordering the sequence of activation. In CIS it consists of being able to activate or deactivate sets of rules. Special purpose rules can easily be added to control the activation of rule sets. It is easier to implement such rules in CIS than in other production systems.

5.12.7 Inexact Reasoning

There are two types of inexact reasoning in CIS. The first concerns forward chaining. Activity values are used for all assertions and when forward chaining, these values are combined by some rules of fuzzy logic. The second concerns selecting questions to ask the user. Activity values on the **want-to-know** nodes of the parameters are used to decide what are the most interesting parameters to ask about. Because the ability to manipulate activity values rather than binary values is built into the primitives, inexact reasoning can be implemented cheaply in CIS.

5.12.8 Timing

In circuit design much effort is spent on timing issues; which signal arrives at a given place first is often important. In CIS timing need not be considered: the relative delays on the links have no effect. In general CIS can be imagined as a large asynchronous circuit with no unstable feedback paths. In the case that contradictory sets of rules are created such as: "if a then b" and "if b then not a", unstable feedback can be created and an oscillation can occur. This is a bug of the rule set and not of CIS and will cause a problem in almost all production systems. To help find these bugs, debugging tools which can recognize oscillations should be supplied.

In other systems implemented in AFL-1 that maintain state within the network (have positive feedback paths), timing issues must be considered. In AFPLAN, discussed in chapter 6, there are a few timing problems but they are easy to get around. In general, a bit to my surprise, I found very few timing problems when programming in AFL-1, but I expect with more complicated systems, timing can become an important issue.

Chapter 6

AFPLAN - A Planning System

Domain-independent planning systems written in the last 15 years such as STRIPS [Fikes71], NOAH [Sacerdoti75], MOLGEN [Stefik81], and TWEAK [Chapman85], have relied heavily on the abstractions supplied by symbol processing languages. This chapter will show how a planner, AFPLAN, achieves much of the same functionality with the abstractions supplied by the activity flow paradigm. Some aspects of the symbolic planners are hard or inefficient to implement with AFNs. These aspects will be discussed. AFPLAN was implemented to explore the potential of activity flow languages; it is not claimed to be a state of the art planner.

Like STRIPS, NOAH, MOLGEN and TWEAK, AFPLAN is a operator/state based system in which the world state is specified by a collection of propositions, and operators are used to change the state by activating or deactivating the propositions. The purpose of the planner is to select an ordered set of operators that when applied to an initial world state results in a goal state. The significant difference among existing planners is the strategy used to select this order.

AFPLAN uses the following method. Between each action, AFPLAN does a complete search from the goal propositions for possible next steps. This search is executed concurrently on a static precompiled network. After the search, a single operator is selected and applied - section 6.3 discusses extensions which allow the selection of more than one operator per step. To be selected, an operator must, a) help satisfy a goal proposition, b) not clobber a proposition needed to achieve another goal, and c) satisfy at least as many goals as any other operator. Strict adherence to these rules can cause deadlock when a side-step is required; Section 6.6 discusses how such deadlock can be avoided.

As in NOAH, AFPLAN detects constraints among branches of the search tree. Like NOAH's *"resolve conflicts"* critic, it detects operators in one branch that delete preconditions needed for operators in another branch. Like NOAH's *"use existing objects"* critic, it notices when one operator can be used to satisfy two goals. Like NOAH's *"eliminate redundant preconditions"* critic, it recognizes when two operators have the same preconditions.

Because of the expense of storing relations, manipulating variables and passing

complex information with activity flow networks (AFNs), AFPLAN and NOAH functionally differ in three important ways.

1. Because it is difficult to manipulate large dynamic structures with AFNs, AFPLAN only manipulates parts of a plan between actions. Systems such as NOAH manipulate the whole plan before taking a single action. Partial planning sometimes leads to non optimal plans but is perhaps a more realistic view of a real world planner. By executing actions between subplans, AFPLAN can take advantage of dynamic feedback from the environment.

2. As with CIS (chapter 5), AFPLAN requires that all variables are quantified over a fixed set of values at compile time. Figure 6.1 shows an example of the definition of propositions and operators in which the variables are quantified over a fixed set of values. The effect of this on the size of network is discussed in section 6.7.

3. Since no pointers can be passed in AFNs, AFPLAN over constrains the network by placing constraints among disjunctive as well as conjunctive subgoals. Section 6.6 discusses this problem and suggests some solutions.

The operator definitions of AFPLAN take six keyword arguments; five are lists of propositions, the *context-list, precondition-list, do-list, add-list* and *delete-list*, and one is a numeric value, the *cost* of the operator. The *precondition-list*, *add-list* and *delete-list* serve the same purpose as they do in STRIPS [Fikes71]. The *context-list* is used for propositions needed as preconditions to the operator that should not be set up as subgoals. The *do-list* can be used to specify functions other than primitive actions to be executed when the operator is applied; section 6.5 discusses this. The **cost** argument is used to specify the cost of certain operators and is discussed in section 6.4.

6.1 An Example

To give a taste of how AFPLAN works, this section describes AFPLAN's solution of the block world's "anomalous situation" [Sussman73, Sacerdoti75, Chapman85]. Figure 6.2 shows the problem; it is termed anomalous because many classical planners such as STRIPS [Fikes71], PLANNER [Hewitt71] and HACKER [Sussman75] made the mistake of putting B on top of C before taking C off A, or of putting C on top of B in the process of clearing the top of A. Since NOAH and AFPLAN consider the interactions among the conjunctive subgoals, they do this problem correctly.

```
(setq block-names '(A B C))

(defstate ((x (in block-names))
           on
           (y (in block-names 'ground (excluding x)))))

(defstate ((x (in block-names 'ground))
           is-clear))

(defoperator (move
              (x (in block-names))
              from
              (y (in block-names 'ground (excluding x)))
              to
              (z (in block-names (excluding x y))))
  :context ((x on y))
  :preconditions ((x is-clear) (z is-clear))
  :add ((x on z) (y is-clear))
  :delete ((z is-clear) (x on y)))

(defoperator (move
              (x (in block-names))
              from
              (y (in block-names (excluding x)))
              to-ground)
  :context ((x on y))
  :preconditions ((x is-clear))
  :add ((x on ground) (y is-clear))
  :delete ((x on y)))
```

Figure 6.1: State and Operator Definitions for the Blocks World.

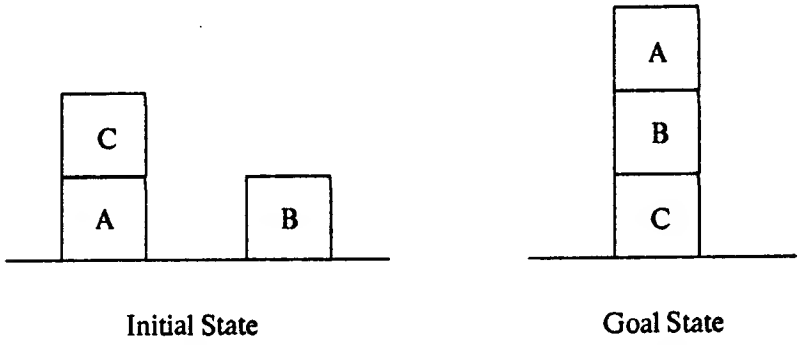


Figure 6.2: The "Anomalous Situation" in the Blocks World.

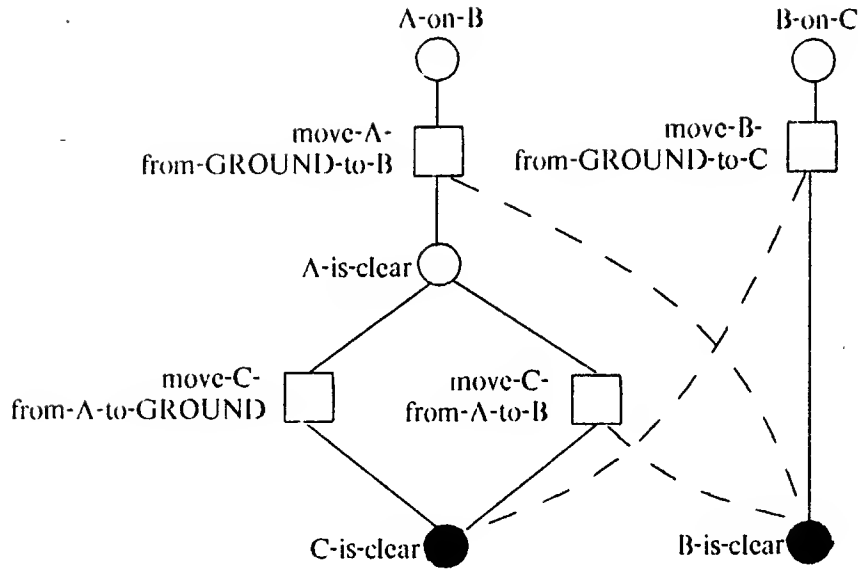


Figure 6.3: The Part of the AFN Activated by the Anomalous Situation.
Dark circles show the active states, and dashed lines show preconditions that get deleted (deactivated) if the operator they are linked to is applied.

The example will use the operators "**move x from y to z**" and "**move x from y to-ground**", and the state propositions "**x on y**" and "**x is-clear**"; figure 6.1 shows their definition. The operators and state descriptions are compiled into an AFN in which the variables are compiled out, so the propositions and operators have a subnetwork (node group instance) for each combination of the variables. Section 6.2 describes the networks that are created in further detail, and section 6.7 discusses the cost of having a separate node group for every proposition.

Once the network is compiled, the user sets up the initial state and specifies the goal state. In the anomalous situation, the propositions of the initial state are **C-on-A, A-on-ground, B-on-ground, C-is-clear** and **B-is-clear,** and those of the goal state are **A-on-B**, **B-on-C**, and **C-on-ground**. Figure 6.3 shows the part of the static AFN that these goals activate.

Since only one copy of each proposition is compiled, branches from different trees share the same nodes (e.g. there is only one **C-is-clear** node).

Three operators in the search tree have their preconditions satisfied. Two of these, **move-C-from-A-to-B** and **move-B-from-ground-to-C**, are the operators applied mistakenly by STRIPS and PLANNER. As the dashed lines in the figure show, both these operators, if applied, will remove a proposition needed by other operators. The AFN detects this interaction and puts the two operators in a blocked state.

Since **move-C-from-A-to-ground** is the only operator left active, the host com-

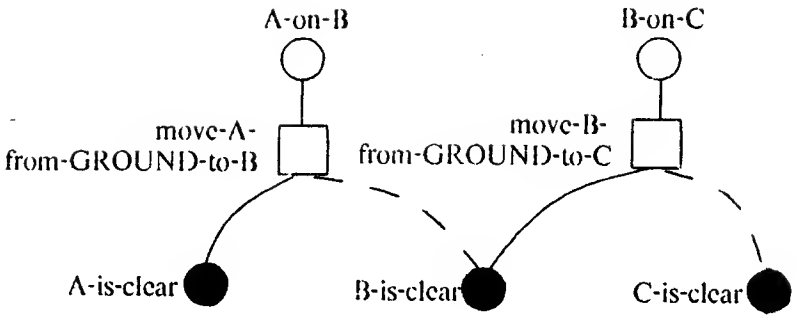


Figure 6.4: The Active Part of the AFN After Moving **C** onto the **Ground**.

puter selects it to be applied (methods of selection are discussed in section 6.3) and sends it a signal which lets it activate the propositions in its *add-list* and deactivate the propositions in its *delete-list*. The propositions that get activated by the **move-C-from-A-to-ground** operator are **A-is-clear** and **C-on-ground**, and the proposition that gets deactivated is **C-on-A**. Figure 6.4 shows the active part of the network after these changes have propagated through the network. In this network **A-is-clear** is blocking **move-A-from-ground-to-B** so the system selects and applies the operator **move-B-from-ground-to-C**. In the final step, the only operator left is **move-A-from-ground-to-B**, so it is selected.

The plan taken by AFPLAN is, therefore: **move-C-from-A-to-ground, move-B-from-ground-to-C, move-A-from-ground-to-B**. This is the best solution. The operators discussed have also been tested on every other problem using 3 or 4 blocks. In all cases, the solution was optimal.

6.2 The Afplan Node Groups

The **state** and **operator** node groups are the most important parts of AFPLAN: these node groups are used to compile the specification of the states and operators into the AFN. In figures 6.3 and 6.4, each of the circles signify an instance of the **state** node group and each of the squares, an instance of the **operator** node group. The only job of the **defstate** and **defoperator** forms is to, at compile time, loop over all the values of the variables and create a node group instance for every combination of these values. At runtime, **input** nodes in the **state** node group are used to set up the goal and initial states and **input** and **output** nodes in the **operator** node group are used to recognize when an operator is ready to be applied and to tell an operator when it has been selected.

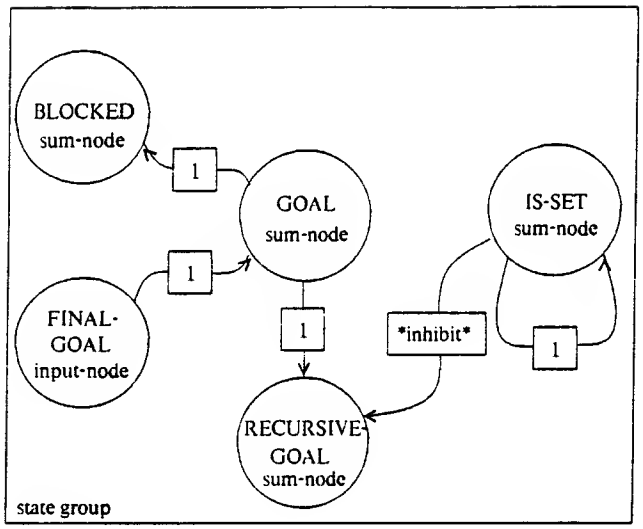


Figure 6.5: The Network of an Instance of the State Node Group.

6.2.1 State Node Group

The **state** node group is instantiated with no arguments. Figure 6.5 shows an instance of the group and figure 6.6 shows the AFL-1 definition. The purpose of the nodes are:

- **is-set** - Is active when the proposition is true.
- **goal** - Is activated by an operator if the operator "wants to be done" and has the proposition as one of its preconditions. It is also activated when the proposition is specified as a final goal by the user.
- **blocked** - Is active when more than one operator has the proposition as one of its preconditions. If this node is active, all operators with the proposition as a member of their delete-lists will be blocked.
- **recursive-goal** - Is active when **goal** is active but **is-set** is not. When this node is active, the **want-to-know** of every operator with the proposition in its add-list will be activated.

6.2.2 Operator Node Group

The **operator** node group is instantiated with six arguments: the *context-list*, *precondition-list*, *do-list*, *add-list* *delete-list*, and *cost*. Figure 6.7 shows the network created by an instance of the **operator** node group and figures 6.8 and 6.9 show the AFL-1 definition of the node group. The purpose of the nodes are:

reminder of syntax:

```
(make-node name &optional (Threshold *active*) (Slope *nil*)
                          (Rise *active*) (Saturation *saturation*))
(make-er-link from-node-name to-node-name &optional (Weight *active*))
(defgroup group-name argument-list &rest body)
```

the state node group:

```
(defgroup state ()
  (make-bistable-node 'is-set)
  (make-node 'goal 1 1 1 *saturation*)

  (make-latched-input-node 'final-goal)
  (make-er-link 'final-goal 'goal)

  (make-node 'recursive-goal)
  (make-er-link 'goal 'recursive-goal)
  (make-ir-link 'is-set 'recursive-goal *saturation*)

  (make-node 'blocked 1 1 0 *saturation*)
  (make-er-link 'goal 'blocked 1))
```

Figure 6.6: The Definition of the State Node Group.

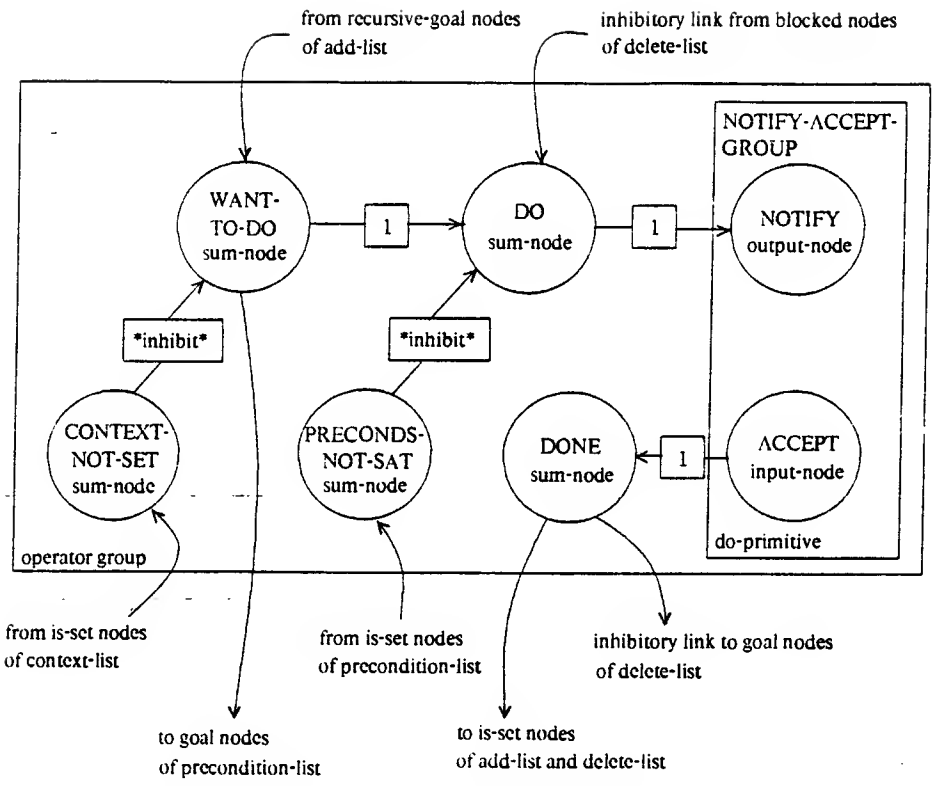


Figure 6.7: The Network of an Instance of the Operator Node Group.

- **want-to-do** - Is activated if one of the elements in the operator's add-list is currently a goal, and all the states in its context-list are active. When the **want-to-do** node of an operator is activated, all its precondition's goal nodes are activated.

- **preconditions-satisfied** - Is activated when all of an operators preconditions are active.

- **do** - Is activated when both the **want-to-do** and **preconditions-satisfied** nodes are active and the **blocked** node is not. The operator is ready to be applied if this node is active.

- **done** - Is activated after the operator is applied. The activation of this node will deactivate all the propositions in the delete-list and activate all the propositions in the add-list.

```
(defgroup operator (&key (context-list nil)
                         (precondition-list nil)
                         (do-list 'primitive)
                         (add-list nil)
                         (delete-list nil)
                         (cost 1))

  (let ((delete-precond-list (intersection delete-list precondition-list))
        (delete-context-list (intersection delete-list context-list)))

    (make-node 'want-to-do 1 1 1)
    (make-node 'preconds-not-satisfied 0 0 1)
    (make-node 'context-not-set 0 0 1)

    (make-node 'do 1 1 1)
    (make-er-link 'want-to-do 'do)
    (make-ir-link 'context-not-set 'want-to-do *inhibit*)
    (make-ir-link 'preconds-not-sat 'do *inhibit*)

    (make-node 'done)
    (make-notify-accept-group 'do-primitive)
    (make-er-link 'do '(do-primitive notify))
    (make-er-link '(do-primitive accept) 'done)
```

Figure 6.8: The Definition of the Operator Node Group (Continued in Next Figure).

```
;; Turn off "context-not-set" node if all contexts are set.
(dolist (context context-list)
  (make-ir-link `(< ,context is-set) 'context-not-set))

;; Set up links from the "want-to-do" node to all the preconditions
;; and from the preconditions to the "preconds-not-sat" node.
(dolist (precondition precondition-list)
  (make-er-link 'want-to-do `(< ,precondition goal))
  (make-ir-link `(< ,precondition is-set) 'preconds-not-sat))

;; Set up links from the "done" node to the elements in the add-list
;; and from the recursive-goal of the done elements to the
;; "want-to-do" node.
(dolist (add-state add-list)
  (make-er-link 'done `(< ,add-state is-set))
  (make-er-link `(< ,add-state recursive-goal) 'want-to-do))

;; Set up inhibitory links from "done" node to the elements of the delete
;; list.
(dolist (delete-state delete-list)
  (make-ir-link 'done `(< ,delete-state is-set))
  (make-ir-link 'done `(< ,delete-state goal)))

;; Used for blocking when element is in both precond and delete list
(dolist (element delete-precond-list)
  (make-ir-link `(< ,element blocked) 'do))

;; Used for blocking when element is in both context and delete list
(dolist (element delete-context-list)
  (make-ir-link `(< ,element goal) 'do))))
```

Figure 6.9: The Definition of the Operator Node Group (Continued from Last Figure).

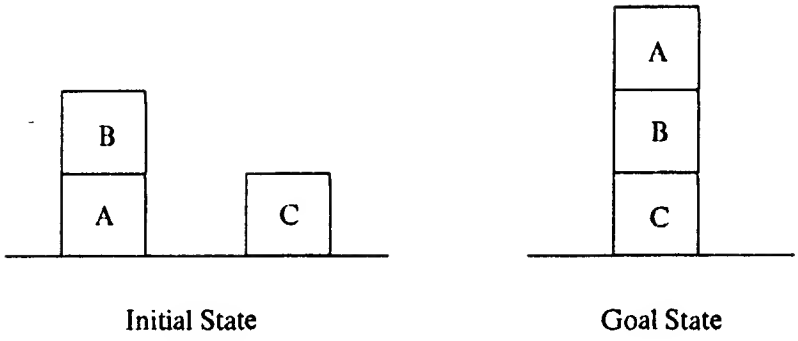


Figure 6.10: Taking the Better Step.

6.3 Selecting An Operator

Because AFPLAN searches for subgoals concurrently, it often puts more than one operator in the ready state simultaneously. There are two potential problems with applying all such operators together: a) the machinery required to execute the operators might be incapable of serving simultaneous requests, and b) if the actions are disjunctive (lead to the same goal), executing all of them would be wasteful.

For example, the goal *"get to the airport"* might set up the two disjunctive subgoals *"drive to the airport"* and *"take the subway to the airport"*. If all the preconditions of these two operators became active simultaneously, and the system tried to apply both of them, there would be problems for both the above reasons.

To prevent these problems, AFPLAN only applies one primitive operator at a time and waits for an accept signal from the outside world before proceeding. To select an operator, AFPLAN finds the operator that will satisfy the most goals if applied. It does this by selecting the operator with the highest activation on its **(do-primitive notify)** node; the activation of this node is proportional to the number of active goals that feed into the **want-to-do** node. For example, in the blocks world problem shown in figure 6.10, moving B to C will satisfy two goals: clearing A and stacking B on C. In this case the **(do-primitive notify)** node of the operator **move-B-from-A-to-C** will be more active than the operator **move-B-from-A-to-ground** and will therefore be selected. Note that, although this uses a completely different method than NOAH's *"use existing objects"* critic, it serves the same purpose.

There are a couple ways for the system to select the operator with the most activated **(do-primitive notify)** node. In the current implementations, the host computer makes the selection. It scans the **(do-primitive notify)** nodes, selects the one with the highest value, and sends a signal to the **(do-primitive accept)** node of the cor-

responding operator. If the number of operators is large, or the AFN has to control machinery directly, then a *winner-take-all* (WTA) network can be used instead (See chapter 7). Such a network would be put over the **do** nodes so that only the one with the strongest inputs will become active. To make this solution work, the implementor will have to take into account the transient effects that happen while the network settles. By using more than one WTA group, and only placing the WTA groups over conflicting operators, the second method could allow non-conflicting operators to be applied simultaneously.

Both the above methods are serialization techniques - they pick a single action to be executed out of a set of potential actions.

6.4 The Operator Cost Argument

It is often unreasonable to assume that the expense of applying different operators is the same. Often one operator requires more energy, takes more time or is more dangerous than another. To allow for these differences, the AFPLAN operator group includes a *cost* argument. When two operators are ready to be applied at the same time, and they both will satisfy the same number of goals, AFPLAN will choose the one with the smaller cost.

Using the mechanism that selects the operator satisfying the most goals (discussed in last section), it is trivial to include this feature. The inverse of the expense given by the user (1 by default) is used as the weight between the **do** node and the **(do-primitive notify)** node, so when the system searches for the operator with the highest activation on the **(do-primitive notify)** node, it will select the less expensive operator. This method is not very sophisticated and surely doesn't consider interactions among the costs of the operators, but could be extended to include more sophisticated techniques.

6.5 The Do List

The *do-list* of the AFPLAN operator can be used to specify three sorts of actions: primitive actions, subgoals, or functions.

Operators are **primitive** by default. Primitive operators send a signal to the outside world when they are executed and expect a signal back when the primitive action is completed. The command "*turn hand 5 degrees clockwise*" might be a primitive action for a robot. A direct connection from the AFN to the machinery that turns the hand and one back to the AFN could implement this action.

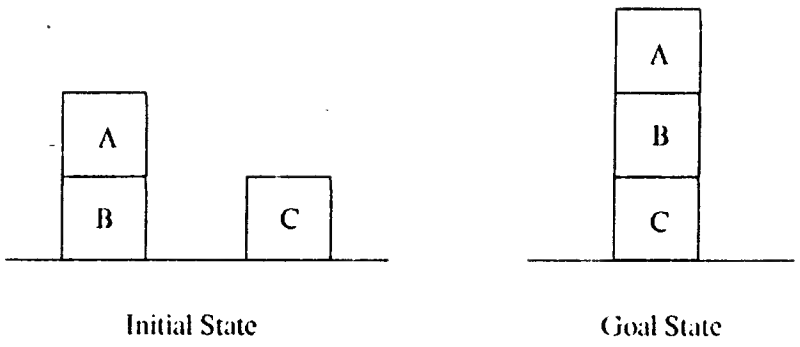


Figure 6.11: The Side-Stepping Blocks Problem.

To allow for the kind of hierarchical planning done by ABSTRIPS [Sacerdoti 74], AFPLAN allows the user to put goals in the *do-list*. This placement of goals has the effect that the goals are only activated when all the preconditions of the operator are satisfied, and the operator is not blocked.

Some actions require more than a single primitive action. For example, the **move-block** operator for a robot is likely to require several primitive actions. The *do-list* of an operator can be used to activate an arbitrarily complex set of actions. This set of actions must be compiled somewhere in the network and is activated through a node specified in the *do-list*.

6.6 Constraints

An instance of a **state** node group (a proposition), uses its **blocked** node to recognize when it is a precondition to more than one operator in the search graph. When the AFN is compiled, an inhibitory link is created from the **blocked** node of each proposition to the **do** node of operators that have that proposition in their *delete-list*. These links prevent the system from *clobbering* [Chapman85] the preconditions of other goal operators.

Although this simple method works in many situations, it has two problems: a) the system can only hill climb and can't take any side or back steps, and b) the system over constrains the network by placing constraints among disjunctive as well as conjunctive subgoals.

If there are no operators that lead further toward the goal, the system described so far will become interlocked. In the blocks world, with three blocks, figure 6.11 shows the only case where this is a problem. To clear B so it can be placed on C, A must be removed from B. This action is a side-step since it requires the undoing of one goal

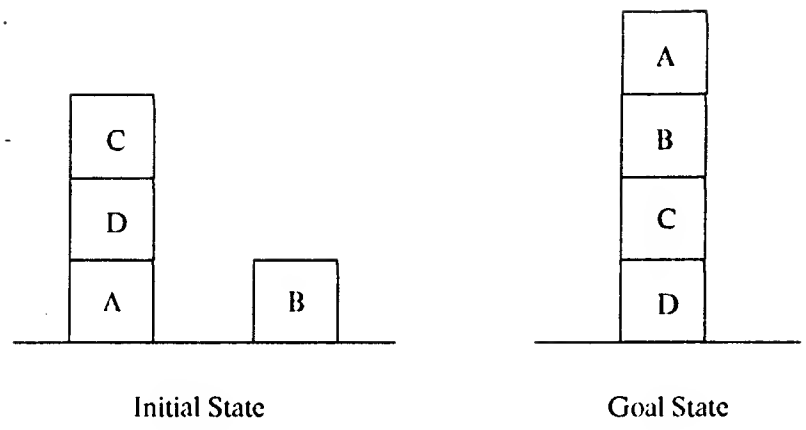


Figure 6.12: The Double Side-Stepping Blocks Problem.

to achieve another. As Phil Agre puts it, it's like getting to the refrigerator door with milk in one hand and orange juice in the other.

To get over this problem, AFPLAN recognizes when it is interlocked by noticing both that no operators are active, and that goals still need to be achieved. When AFPLAN recognizes it is interlocked, it raises the threshold on all the **locked** nodes by supplying a controlled negative input to those nodes. This will allow the application of operators that only delete a single precondition. Although this method works with the three blocks world, it is not a general solution because it doesn't address the problem of selecting which side step is least destructive. For example, figure 6.12 shows a problem in which two side steps can be taken. The system can either put **B** on **C**, or **C** on the **ground**; both achieve one goal at the expense of another.

AFPLAN happens to take the right step in this problem but only because the operator "**move x from y to z**" is set with a larger cost than "**move x from y to-ground**".

The second problem with the blocking method is that it over constrains the plan; in particular it can cause disjunctive as well as conjunctive goals to block each other. This is rarely a problem in the blocks world because disjunctive goals rarely interact, but in other worlds it can be a serious problem. This over constraint is inherent in the system and can only be overcome by searching a single disjunctive goal at a time. In the network generated by AFPLAN, there is no way of telling at a state whether two signals are coming from conjunctive or disjunctive subgoals. The problem is related to the "find everybody who is their own father" problem mentioned in [Hillis81] and [Fahlman83] and a node limited message state network (discussed in chapter 2) is required to solve it. Fahlman called such networks "painted tokens" networks.

Another problem AFPLAN has with disjunctive subgoals is that once the system starts applying operators in one disjunctive subgoal, nothing deactivates the search for operators in other branches of the disjunctive tree. This problem is a problem with AFPLAN and, unlike the previous problem, is not due to any limitation imposed by AFNs. There is a relatively easy fix that prevents the problem. This fix requires an extra signal that travels up the tree of goals. When an operator is applied, it activates this signal, and the signal travels up the tree turning off branches of disjunctive subgoals as it passes.

6.7 Network Required

An obvious question to ask is, how much network (how many links) is required to implement a plan in AFPLAN? In particular, if a variable can have many values, won't a large amount of network be required to include an **operator** or **state** node group instance for every permutation of the variable values?

The three blocks problem shown in figure 6.1 requires 377 links, the same problem with four blocks requires 957 links. Neither of these are coming close to the practical limit of 1-4 million links discussed in chapter 8. A blocks planner could plan with up to 200-300 blocks within the practical limit or today's computers, and probably with 10,000 blocks within the next ten years. Considering that humans will have difficulty planning with more than 10 blocks, these limits are certainly not unreasonable.

Although the network compiled for the blocks world used the three specific blocks, **A**, **B** and **C**, the same networks could be used for stacking any sorts of objects. Instead of **A**, **B** and **C**, we could have used **this-object**, **that-object** and **the-other-object** when we compiled the network. Then at run time when we come across three objects to stack, we somehow bind each of those objects to to the **this-object**, **that-object** and **the-other-object**. This binding can either be done by the host, or internal to the network.

6.8 Conclusion

AFPLAN shows how a planner can be implemented on static concurrent networks with AFL-1. The planner has many of the capabilities of previous planners but has a couple limitations. One limitation that seems to be inherent in the AFN model is that it is hard to concurrently search for disjunctive and conjunctive subgoals without either under-constraining the conjunctive subgoals or over-constraining the disjunctive subgoals.

Chapter 7

Mutually Exclusive Groups

A powerful abstraction of activity flow networks and, more generally of connectionist networks, is the mutually exclusive group of nodes. As one element is further activated in such a group, the other elements tend to be inhibited. Some form of such groups can be found in the majority of work on connectionist models. In general, mutually exclusive groups can be cheaply implemented, and can take full advantage of concurrency. This chapter discusses several types of mutually exclusive groups, shows how they can be cheaply implemented, and discusses where they can be used.

In a group of mutually exclusive nodes, the element, or sometimes elements, with the strongest input stay active while the elements with lower inputs are inhibited. As inputs, mutually exclusive groups can use information from many different sources. Figure 7.1 shows an example of an mutually exclusive group. The group uses color, shape and size as inputs to decide on whether the fruit is an apple, banana or strawberry. Using this information, the group will settle on one of the three fruits. Because of their properties, mutually exclusive groups present a good mechanism for interpreting input, focusing on a limited set of things, or arbitrating for a limited resource.

Many recent systems developed using static finite message state (FMS) networks use mutually exclusive groups extensively. In vision systems, they are used to select characters [Rumelhart81], to select a frame of reference [Hinton85], and to interpret visual features [Feldman85]. In natural language systems, they are used to choose the syntactic role [Cottrell85] and semantic meaning [Waltz85] of a word, recognize the syntactic form of a sentence [Selman85], and select words for language production [Dell85]. In logical reasoning systems, they are used to select a matching for a unification algorithm [Ballard84], and to choose among clauses and rules in an inference system [Touretzky85]. For semantic network systems, they are used to decide on the type of an object [Shastri85]. Minsky [1986] uses such groups as a large part of his theory of mind.

Some of these systems use distributed representations [Touretzky85]. With distributed representations, each object, such as **apple**, is represented by a pattern of activity in the network rather than by a single node. In these systems, mutually exclusive groups do not select a single node but rather are only stable in patterns that

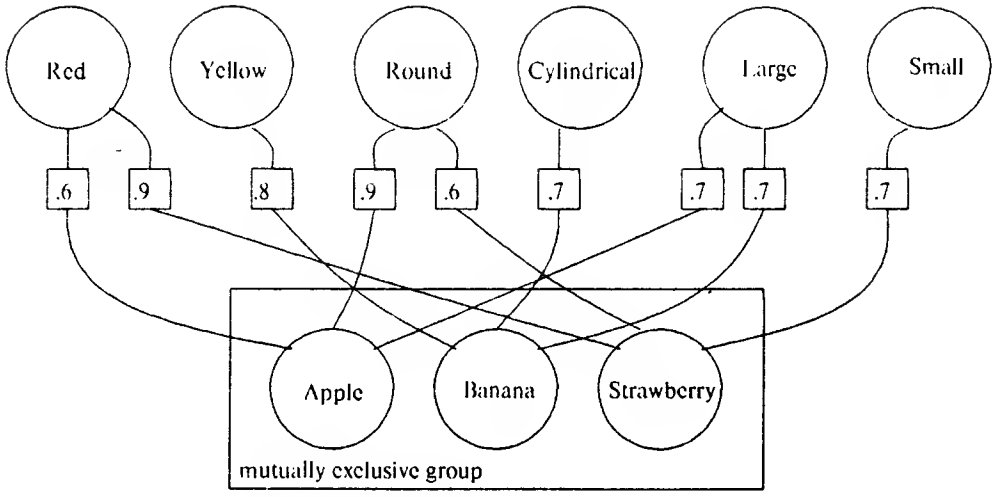


Figure 7.1: An Example of a Mutually Exclusive Group.

represent a single object.

Although the intuitions behind mutually exclusive groups are well understood, their use is often haphazard and the suggested implementations often impractical. Many different types are being used, each with a different function; the differences in function include whether the elements completely inhibit each other, and whether there is hysteresis. The implementations discussed often suggest that an inhibitory link is used between every pair of nodes of the collection; such implementations requires N^2 links and are therefore impractical for large collections.

In AFL-1, mutually exclusive groups can be implemented with the **group** abstraction introduced in chapter 4; groups of this kind will be called ME-groups. This chapter defines five different types of ME-groups and shows how each can be implemented in AFL-1. It then discusses four ways in which they can be used.

7.1 Five Flavors Of Mutual Exclusion

This section categorizes ME-groups into five flavors: **winner-take-all** groups, **winner-take-all** groups with **hysteresis**, **contrast-enhancement** groups, **controlled contrast-enhancement** groups, and **controlled contrast-enhancement** groups with **fatigue**. Each of these groups is useful for a slightly different purpose so they should all be present in an activity flow programming environment. Sections 7.1.1 through 7.1.5 define the flavors and for each flavor, show an implementation using the AFL-1 primitives of a two member group. Section 7.1.6 shows how n element ME-groups of any of the flavors can be implemented using $O(n)$ links.

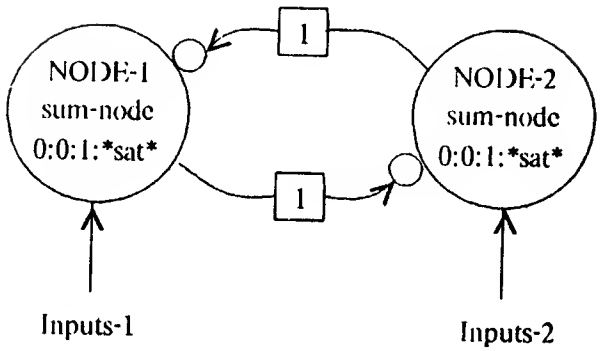


Figure 7.2: Implementation of a Two Member Type 1 ME-Group.

7.1.1 TYPE 1: Winner Take All

DEFINITION

A **winner-take-all** group activates the member with the highest input and completely inhibits all the other members. If at any time another element starts receiving a greater input than the current winner, it will become active and inhibit the other nodes (becomes the winner).

IMPLEMENTATION

Figure 7.2 shows an implementation using the AFL-1 primitives of a two element **winner-take-all** group.

With this implementation, the **winner-take-all** group suffers a time delay when switching between one winner and another. The delay is proportional to the difference in input activity; both nodes are partially active during the delay. An advantage of such a *toggling delay* is that random noise or transients will get damped out.

A disadvantage of the given implementation is that when both elements are off, and receive the same input at the same time, they get stuck in an oscillating state. One way to solve this problem is to split the inhibitive link in two, each with a weight of 1/2, and delay one of the branches by a cycle. This method requires two extra nodes and four extra links and means that in the case of equal inputs, there will be equal outputs.

USES

Winner-take-all groups are used when a single member has to be selected from a group of contenders and all the other members have to be completely inhibited. Such an effect is useful when arbitrating the use of a limited resource.

For example, a **winner-take-all** group might be useful for controlling a heuristically driven elevator. Such an elevator could be designed to have three commands to control its motion, *stop, move-up* and *move-down*; a **winner-take-all** group would be wrapped around these commands since the elevator can only obey one at a time. A set of heuristic rules could contribute as inputs to this group. For a multi-elevator building, the heuristic rules might include:

- Keep the elevators on different floors.
- Spread elevators evenly over the floors.
- Keep one elevator on the ground floor.
- Give priority to the direction where more people are waiting.
- Don't stop at too many floors.
- Keep as many elevators going up as coming down.

Using the heuristic rules as input, an ME-group for each elevator would select one of the commands *stop, move-up* or *move-down.*

People use some sort of **winner-take-all** process to select a single word when speaking. If one were to make the muscle movements for several different words, the result would be incomprehensible.

Winner-take-all networks have the property that they easily change there decision when two elements have similar inputs. This property is sometimes undesirable since it might cause an elevator to get stuck fluctuating between two floors, or cause people to change the word they are saying half way through. To keep an ME-group stable in face of ambiguous inputs, the group can be implemented with hysteresis.

7.1.2 TYPE 2: Winner-Take-All With Hysteresis

DEFINITION

A **winner-take-all** group with **hysteresis** is a **winner-take-all** group with the property that a node can only become the winner by receiving an input of some thresh-

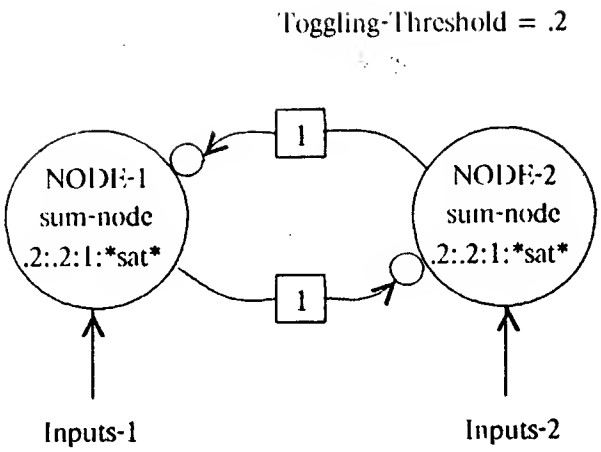


Figure 7.3: Implementation of a Two Member Type 2 ME-Group.

old above the current winner's input. This threshold will be called the **toggling-threshold**.

IMPLEMENTATION

Figure 7.3 shows an AFL-1 implementation of a **winner-take-all** group with **hysteresis**. The **toggling-threshold** is set, and is equal to, the threshold on the individual nodes. By making the threshold different on the two nodes, one can make an asymmetric **toggling-threshold**. This asymmetry can be used to create ME-groups in which some states are more stable than others.

USES

The hysteresis of the TYPE-2 ME-group is useful when a stable and discrete decision has to be made. Many of us have come to an exit on a highway without knowing whether to exit. Those of us with little hysteresis in decision making are the ones who have ended up in the grass between the exit and the highway.

7.1.3 TYPE 3: Contrast Enhancement

DEFINITION

Unlike a **winner-take-all** group, a **contrast-enhancement** group does not completely inhibit the losers but rather just enhances the contrast among the activations of the members. This means the activity ratio two members of the group will be greater than the input ratio of those members. The **contrast-factor** specifies the amount the

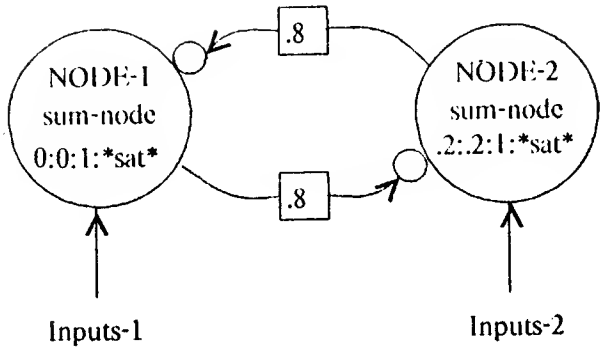


Figure 7.4: Implementation of a Two Member Type 3 ME-Group.

contrast ratio is increased by.

IMPLEMENTATION

Figure 7.4 shows an AFL-1 implementation of a **contrast-enhancement** group. In this implementation the **contrast-factor** is set by the weights on the links. By increasing this weight, the contrast is increased. A **contrast-factor** of 1 implements a **winner-take-all** network; a **contrast-factor** of 0 will create no inhibition between the members. For this implementation, the following equation gives the output in terms of the inputs and the **contrast-factor**.

```
Output = (Input - contrast-factor * Other-input) /
            (1 - contrast-factor**2)
```

Figure 7.5 shows the ratio of the outputs of two members in terms of their input ratio for three different **contrast-factors**.

USES

Contrast-enhancement groups have applications in problems that require mutual exclusion but do not require that the losers be completely inhibited. Applications requiring recognition or interpretation internal to the system can usually take advantage of **contrast-enhancement** groups to allow for partial activation of several different interpretations. For example, your home robot might be programmed to pick up your paper. When it goes to the door mat and reaches down and feels a round object, it probably does not want to immediately settle on the interpretation that it is your paper

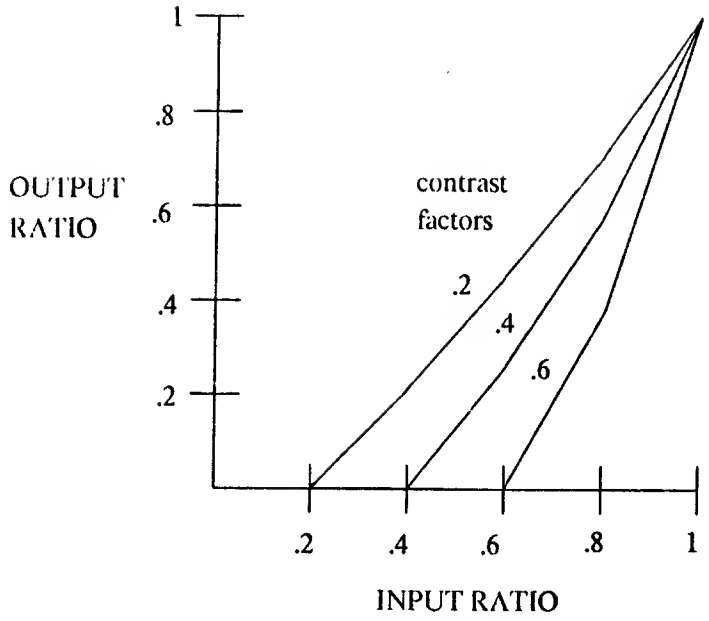


Figure 7.5: Graph of the Contrast Enhancement of a Type-3 ME-Group.

and inhibit all other possibilities. If it brought you a screaming cat, you might not be to happy. In natural language understanding. contrast enhancement might be helpful since the information needed to disambiguate a word often comes after the word itself. In this case all interpretations should stay partially active.

7.1.4 TYPE 4: Controlled Contrast Enhancement

DEFINITION

A **controlled contrast-enhancement** group is a **contrast-enhancement** group in which the **contrast-factor** is controllable from within the network.

IMPLEMENTATION

Since the AFL-1 primitives do not support multiplication of signals, the implementation of **controlled contrast-enhancement** groups is expensive. To implement them efficiently, an extra node type that multiplies as its combining function, can be added (see section 3.8). Figure 7.6 shows an implementation using the multiplication nodes. This implementation has the same properties as the implementation of the fixed **contrast-factor, contrast-enhancement** group, except that the **contrast-factor** can now be controlled by the **cf-input**. This signal can come from anywhere in the network.

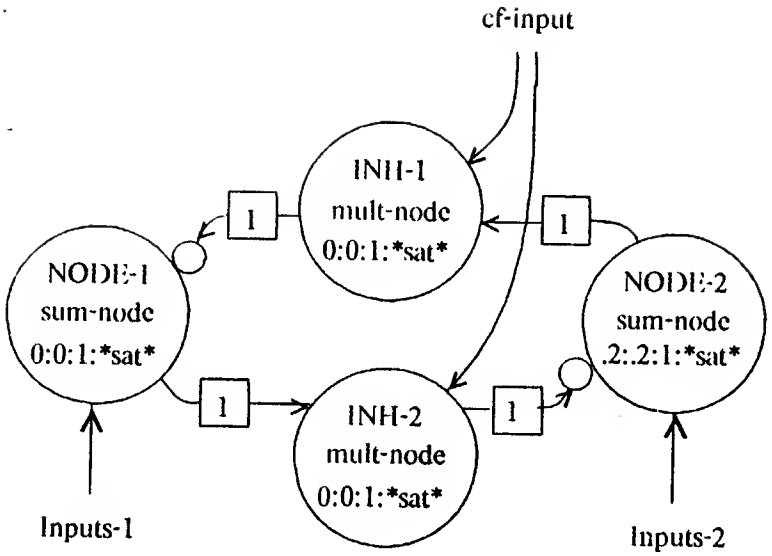


Figure 7.6: Implementation of a Two Member Type 4 ME-Group.

USES

In practice it is useful to initially allow many conflicting interpretations so all possibilities can get a fair footing, and as the computation proceeds, to exclude all but one interpretation. The controlled **contrast-factor** can be used to achieve this effect; early in an interpretation the **contrast-factor** can be kept low and then increased either when the system decides it needs a single interpretation or when the interpretations become stable. Such a mechanism is used in [Hinton85b] for character recognition - he calls it a *competition function* and uses a *schedule* of increasing competition to settle down onto a single interpretation without making serious errors. Similar methods are used in the work on simulated annealing [Hopfield82, Kirpatrick83]. The difference between these methods and controlled **contrast-enhancement** groups is that the later are controlled from within the network while the former are controlled from outside.

7.1.5 TYPE 5: Controlled Contrast Enhancement With Fatigue

DEFINITION

The final type is a **controlled contrast-enhancement** group in which the members with high activation fatigue over time. In such a group, no winner (member with the highest input) will stay a winner for a long time.

IMPLEMENTATION

Such a mechanism is again expensive to implement using the AFL-1 primitives but would be cheap with some extra capabilities built into the primitives. If *fatigue* was built into the primitives then the implementation could be the same as shown in the last section with some fatigue parameter on the node. To implement fatigue the primitives would have to maintain state so they can keep a measure of how long they have been active.

USES

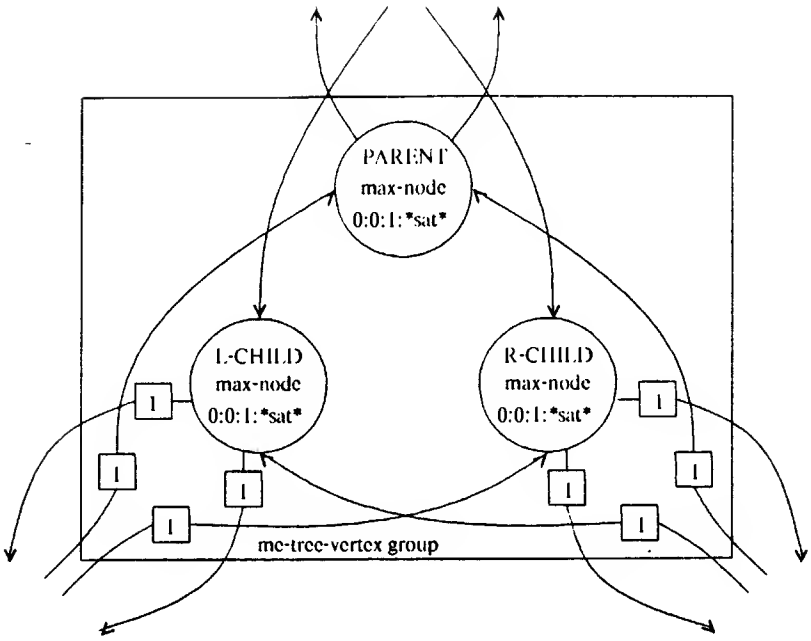
Fatiguing is used to prevent systems from locking up. In single settling systems of the sort described in [Hopfield82] and [Ackley85], the network is only needed to settle into a single interpretation. An external force is then used to reset the network to prepare it for the next interpretation. In contrast, self supporting systems with no external control must be able to reset themselves; one possible way of doing this is by using ME-groups that fatigue. In such a system, any settling will only be stable for some amount of time. This time could be set as a parameter when programming the network.

7.1.6 Implementing Large ME-Groups

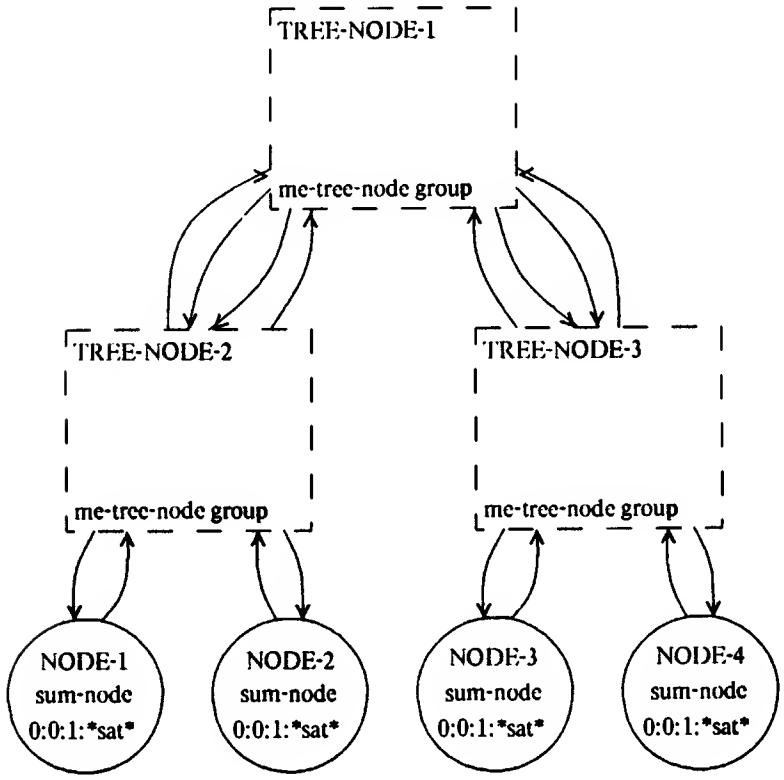
For ME-groups to be practical, the number of links they require must be linear or close to linear in the number of members of the group. This section describes an implementation that can be used with slight variations for all the flavors of ME-groups described above. For groups with N members, this implementation requires $7N$ links and $3N$ nodes. A similar $O(N)$ network is discussed in [Koch84], but this network can only be used for **winner-take-all** groups.

The implementation is constructed using a tree structure in which the group of three nodes shown in figure 7.7a is used at each vertex of the tree. This group of three nodes has the property that the output activity along any of the three directions is equal to the maximum of the inputs along the other two directions. Figure 7.7b shows a four member **winner-take-all** network, built with such a tree. The signal received at each leaf of the tree is the maximum activation of all the leaves of the tree excluding itself. This can be proved inductively by showing that it is true for a tree of two leaves, and that if it is true for a tree of 2^n leaves then it is true for a tree of 2^{n+1} leaves by putting two subtrees together.

Since at each member of the ME-group, the maximum of the other members is available, each member can treat that signal as if it came from a single other element.



A)



B)

Figure 7.7: Implementation of Large ME-groups.

This implementation as $N \rightarrow \infty$ requires $7N$ links and $3N$ nodes. Several other $O(N)$ implementations exist.

7.2 Uses Of ME-Groups

Since there is not a one to one map between types of ME-groups and their uses, this section summarizes the uses. The uses are broken into four categories, output serialization, internal selection, internal serialization and buffers.

7.2.1 Output Serialization

Often an output, or other resource, of a concurrent system can only process a single request at a time. If many parts of the system ask for this resource at the same time, the requests have to be serialized. Two examples of such outputs were discussed in the uses of **winner-take-all** groups, one for an elevator controller and one for natural language production. Since the resource of concern can only process a single request, the ME-group used must be some sort of **winner-take-all** group.

In the Concurrent Inference System (Chapter 5), serialized output is used for question asking. The section on focus (5.10) discusses how an ME-group can be placed over all the **ask** nodes so that only one is asked at a time. In AFPLAN (Chapter 6), serialized output can be used to prevent simultaneous commands from being sent to a device that can only perform a single action at a time.

7.2.2 Internal Selection (Interpretation)

People have a remarkable ability to select an interpretation from a large group of ambiguous possibilities when aided by a large amount of contextual information. One way to implement the part of the interpretation system that chooses which interpretation is best might be to place ME-groups around conflicting interpretations. If the interpretation is internal to the system, the system does not have to completely inhibit secondary interpretations. By internal I mean that no external commitment, such as muscle movement, is directly controlled by the interpretation. Because of this, the implementation can use a **contrast-enhancement** group.

A common use of such interpretation ME-groups is to select a single value of a parameter/value binding. For example, one may want to decide if a ball is red, blue, yellow or green; whether the day is Monday, Tuesday, ... , Sunday; or whether the character is 'a', 'b', ... 'z'. The Concurrent Inference System (CIS) uses such selection

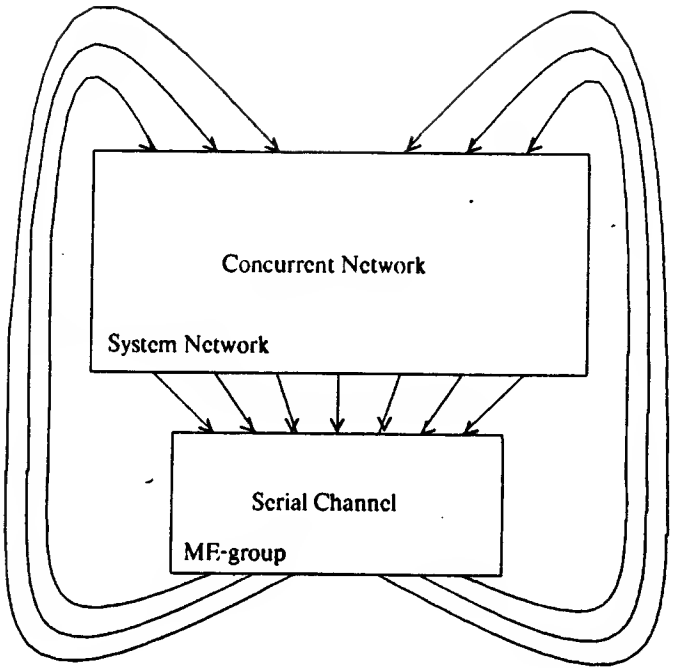


Figure 7.8: The Mechanism of Internal Serialization.

to chose a value for each parameter (see section 5.5). Some of the Focusing techniques proposed for CIS also use internal ME-groups.

7.2.3 Internal Serialization (Attention)

Many researchers agree that even though the brain process must have significant concurrency, there seems to be a serial aspect to the highest levels of thought [Newell72, Treisman82]. This part of thought is often called attention. How can the concurrent architecture of the brain support this apparently serial processing? One possible way is to use some sort of serial path in which whatever is presently "in attention" has to go through - such serialization will be called internal serialization.

Implementationally, internal serialization can consists of selecting a single member from a large group, passing the selected element through a single serial path, and using the output of the path to feed back into the system. A diagram of this mechanism is shown in figure 7.8. What Vitosky calls *private talk* - the words we say to ourselves when we think - might be supported by some form of internal serialization. Like external serialization, internal serialization can be used to allocate a resource, but instead of being an external resource, it is the network of the brain itself.

There are two important differences between internal serialization and internal selection. First, in internal serialization there is only one group (path) in the whole

system, while in selection there can be many selection groups. As a consequence, the internal serialization group is very large while the selection groups are usually small. Second, internal serialization is used for high level functions to be executed serially while selection is usually used for low level functions - interpretations.

The inference mechanism of Touretzky [85] uses a form of internal serialization. It forces only a single clause to be active in each of two clause groups. These clause groups are in effect a serial path. The **word-society, pro-neme, k-line, polyneme, pre-neme, word-society** loop discussed by Minsky [Minsky86] also uses a similar mechanism.

7.2.4 Registers And Buffers

A register is a slot into which one of a finite set of codes or symbols can be put. ME-groups also have this property: they allow one of a finite set of nodes to be active at a time. A register made from an ME-group can be more flexible than a conventional register since it can allow partial activation of more than one symbol, can easily increase in size as more elements are added to the group, and can vary in size within the system so that each register is only as big as it has to be.

A chain of the ME-group registers can be used as a buffer. Such a buffer could be used as the feedback path discussed in the section on internal serialization. It can also be used to buffer the last few sounds one hears, or some information about the last few visual inputs. Pass gates can be used to move the values through the ME-group buffer. The implementation of such a buffer is simple.

7.3 Problems With ME-Groups

Although ME-groups are useful abstractions they do not seem to accurately model the general interactions among thoughts and interpretations of the human brain. Like any modularization technique, ME-groups impose symmetries and boundaries which are not natural. For complex systems, it becomes hard for the implementor to decide which concepts belong in which group, and what groups to use. It also becomes hard to take into account the interaction among ME-groups.

Chapter 8

Implementation

This chapter discusses issues involved with implementing an AFL-1 **network-processor**. It describes the current implementation, discusses general issues of implementing large fine-grained static networks, and suggests better hardware for such networks. Before the current implementation on the Connection Machine (CM) is discussed, section 1.1 gives a short outline of the CM as background, and section 1.2 describes a conceptual implementation.

The implementations brings out two general issues regarding the design of finite message state (FMS) networks (see section 2.2 for definition). Firstly, the cost of large networks is a function of the number of links rather than the number of nodes. Secondly, links with some locality (links between nodes that are close) are much cheaper than random links. Many researchers have counted nodes instead of links in their analysis of the cost of FMS networks [Feldman81, Touretzky85], and it is unclear that their conclusion are still valid if links are counted instead. Section 1.5 discusses some of these issues.

Although the CM is good for experimenting with AFLs, it is more general and therefore costly (time, space and price) than needed. In particular, it is hard to take advantage of local communications on the CM. Section 1.6 suggests some changes that can be made to the hardware so that larger less expensive networks can be implemented.

In AFL-1 the efficiency of the **network-processor** is defined in terms of the number of values passed through the links per second (LIPS). This measure is calculated by dividing the number of links in the network by the time taken by an **afl-step** (see section 3.5). Table 1.1 shows an estimate on the maximum number of LIPS for various computers.

8.1 The Connection Machine

This section gives a brief overview of the Connection Machine (CM); a more detailed description can be found in [Hillis85], and [Christman84].

- Processors - The CM has a large number of very simple bit serial processors. The current Thinking Machines Corporation version has 64K processors and the pro-

IMPLEMENTATION	RATE
Symbolics 3600	.03 mega-LIPS
Vax-780	.1 mega-LIPS
Cray-1	5 mega-LIPS
Connection Machine	75 mega-LIPS

Table 8.1: Approximate Number of LIPS for Various Computers.

OPERATION	RUNNING TIME
Single Bit Logical Operation	1 time unit
32 Bit Floating Point Multiply	1000 time units
Sending a 32 Bit Message From Every Processor	800 time units
The AFL-1 Node Function	100 time units
The AFL-1 Binary Combining Functions	10 time units

Table 8.2: Relative Running Times for Various CM Functions.

posed General Electric/Massachusetts Institute of Technology version will have 256K processors.

- Memory - Every processor has approximately 10^4 bits of its own memory.
- Instructions - The CM is a single instruction multiple data machine (SIMD). Each processor only listens to the instruction stream if its **context-flag** is set.
- Communication - There is a general routing mechanism which allows every processor to send data to any other processor by having or computing a pointer to the other processor.

Table 1.2 shows the relative running times for various functions on the TMC Connection Machine.

8.2 Simple Conceptual Implementation

Although the following implementation is inefficient, it is a good way of picturing how an AFN works since it is simple to understand and does not dependent on the

specifics of the Connection Machine. The implementation is similar to those suggested in [Fahlman79], [Christman84] and [Hillis85].

Each processing element is one of four types: a node processor, a link processor, a fan-in processor or a fan-out processor.

Each node processors is connected to (has a pointer to) a single fan-out processor. The fan-out processors have pointers to two other processors which are either other fan-outs or links. When a fan-out processor receives a value, it performs the spreading function on it (a simple copy in AFL-1), and sends it to the two outputs. By using the tree of fan-out processors, a single node can spread its value to an arbitrary number of link processors. This tree can be kept balanced by the compiler.

When the nodes send out their values, these values propagate through the fan-out tree until they reach a link processor at which point they stop and wait. When all links have received their inputs, they all perform the internal-link function (multiply in AFL-1), and send the result to their output: a fan-in processor. Each fan-in processor has two inputs and a single output. The fan-in processors wait until they have a value in both of their input boxes, run the combining function on the pair of values (MIN, MAX or SUM in AFL-1), and send the result to their output.

When the links send out their values, these values propagate through the fan-in trees getting combined along the way until they reach a node processor. When all node processors have received their values they perform the node-function on this value and their internal parameters (Threshold, Slope, Rise and Saturation) to derive their output value. The cycle is repeated for each **afl-step**. Figure 1.1 shows an example of the layout of the processors.

The problem with using this implementation is that it requires several message cycles for every **afl-step**. If some nodes have large fan-ins or fan-outs, the number of message cycles can be large and the nodes with small fans spend most of their time doing nothing. The next section describes an implementation that overcomes these problems.

8.3 AFL-1 Run Time Implementation

Sending messages on the CM is expensive. This section describes an implementation of AFL-1, which by arranging the links so that it can take advantage of a class of algorithms called **scans**, only requires a single routing cycle per **afl-step**. Unlike the previous implementation the time required by an **afl-step** is independent of the size of the largest fan-tree. Before proceeding, this section will discuss how the CM can

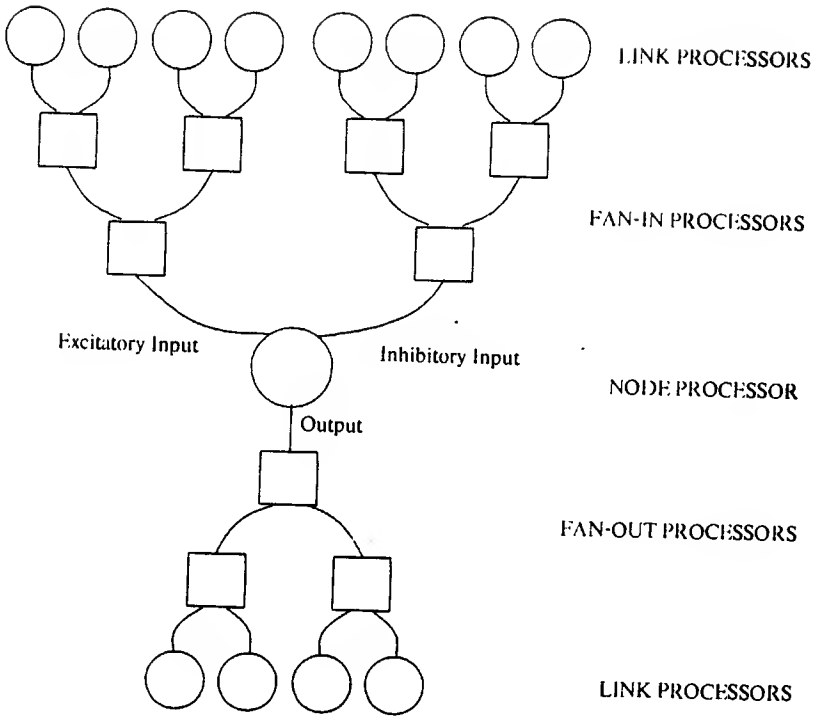


Figure 8.1: An Example Layout for the Simple Implementation of AFNs on the Connection Machine.

simulate more processors than actually exist, and will describe the **scan** algorithms. With a few changes, the implementation described can be used for almost any finite message state network.

8.3.1 Virtual Processors

Since the number of links in an AFN might be larger than the number of processors in the CM, there must be some way to include more links than processors. The CM supplies an abstraction called the *virtual processor* (VP) for this purpose. A VP rather than being an actual processor is a piece of memory within a processor. There can be a large number of VPs in the memory of each processor. When a operation is sent to the CM at run time, the physical processors loop over each of the VPs executing the operation. For operations on local memory (within a processor), this will cause a linear slow-down of execution time with the number of VPs. For operations which require communications, this slow-down is generally non-linear.

8.3.2 Scan Operations

The scan operation, sometimes called initial prefix, takes a binary associative oper-

Processor Number	:	0	1	2	3	4	5	6	7	8
Data	:	1	4	6	3	1	2	5	2	4
Result of Sum Scan	:	1	5	11	14	15	17	22	24	28

Figure 8.2: Example of Scan Using **Sum**.

ator $\otimes$ and an ordered set $A_0, A_1, ..., A_{n-1}$ of n elements, and returns the ordered set $A_0, (A_0 \otimes A_1), ..., (A_0 \otimes A_1 \otimes ... \otimes A_{n-1})$. Figure 1.2 shows an example of the result of the scan operation using the **sum** operator.

Scans run in $O(\log n)$ time on n processors connected in a tree [Schwartz80, Fitch83] and since there are trees embedded in the CM, this bound is valid for the CM. If VPs are used one can show that the required time is $O(r + \log p)$ where r is the VP ratio (the number of VPs per physical processor), and p is the number of physical processors [Kruskal85]. This means that a scan with a $\log p$ VP ratio only requires about twice as long as a scan with a VP ration of 1.

On the CM, scans using simple operators such as **or, and, integer sum,** and **integer maximum** typically run faster than a message cycle. This is because the CM is not a complete hypercube or butterfly but is a complete tree. As the number of VPs increases, the ratio of running times gets larger since the router slows down worse than linearly while scans slow down with a small linear constant.

Segmented Scans

The AFL-1 runtime routine uses a flavor of scans called segmented scans. These scans work on segments of the CM. A segment is a set of adjacent processors whose first processor is marked by some bit in its memory. Figure 1.3 shows an example of 8 processors broken into three segments.

Segment scans only scan over the processors within their own segment. Since they are a special case of general scans they still only require $O(\log n)$ time. The AFL-1 runtime routine uses four operators in its segment scans, **addition, maximum, minimum** and **copy.** Figure 1.3 shows the result of these scans on some example data.

```
Processor Number :   0  1  2  3  4  5  6  7  8

Segment Flag     :   1  0  0  1  0  0  0  1  0

Data             :   1  4  6  3  1  2  5  2  4

Sum  Scan        :   1  5 11  3  4  6 11  2  6
Max  Scan        :   1  4  6  3  3  3  5  2  4
Min  Scan        :   1  1  1  3  1  1  1  2  2
Copy Scan        :   1  1  1  3  3  3  3  2  2
```

Figure 8.3: The Scan Functions Used By AFL-1.

8.3.3 The Implementation

To take advantage of the scan operations, the AFN is laid out on the CM as shown in figure 1.4. Within the order of the CM processors, each AFL-1 node follows all of its input links and is followed by all of its output links.

The output links each have their **weights** and a pointer to the other end of the link (one of the input links of another processor). An **afl-step** then requires the following simple routine:

```
all output links : Segmented Copy Scan the node output-value.
                   Do the link function.
                   Send result to other end of link.
all input links  : Segmented Sum Scan.
                   Segmented Max Scan.
                   Segmented Min Scan.
all nodes        : Do the node function.
```

This routine consists of four scans (it can be reduced to three by doing MAX and MIN together), one message cycle, and the node and link functions. Figure 1.5 shows the running time of the routine for several VP ratios.

For many tasks, with the running times shown, it is practical to have up to 4 million links in a network. With 4 million links, the system could still run 10 **afl-steps** per second.

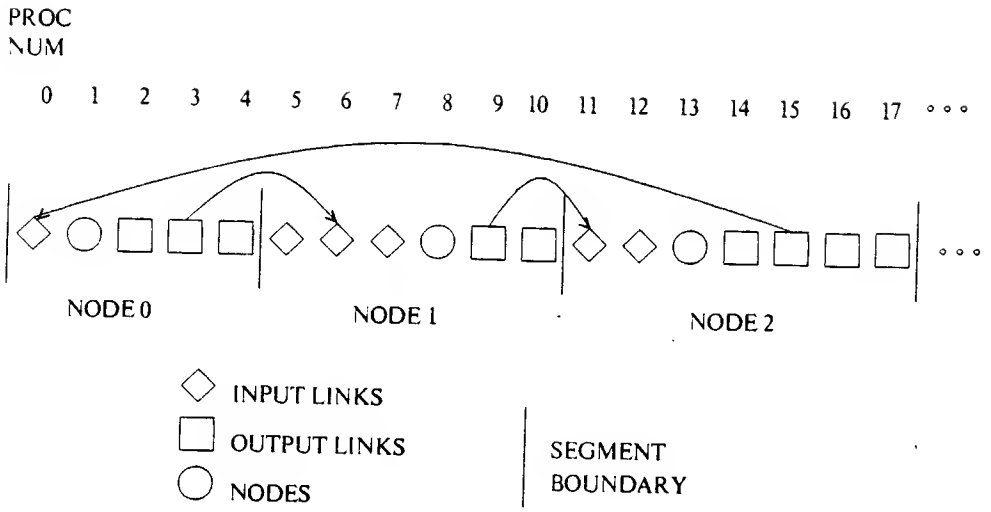


Figure 8.4: Node and Link Layout on the Processors.

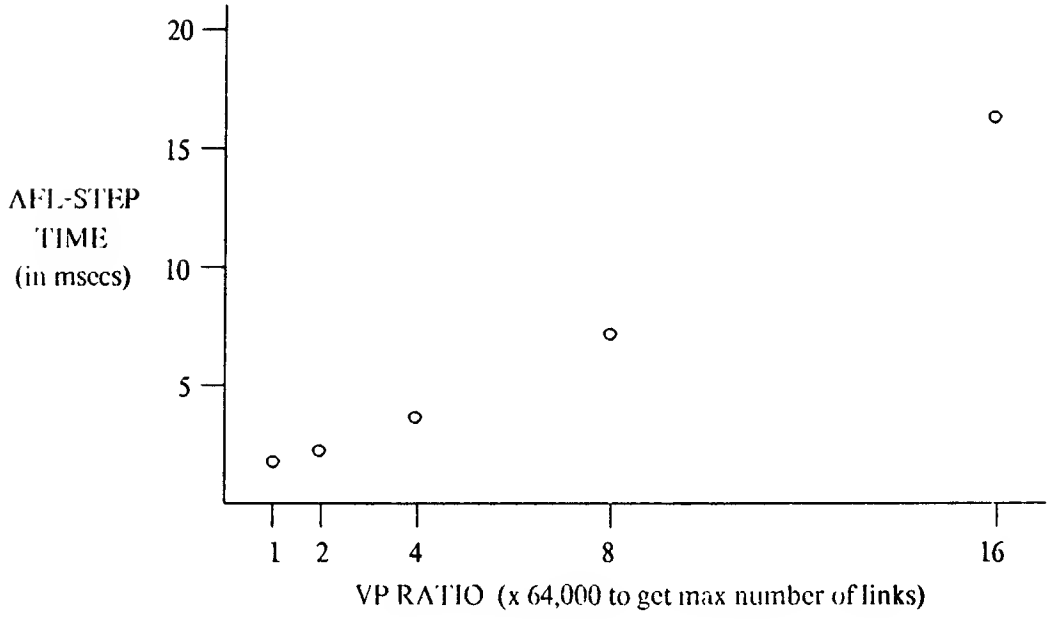


Figure 8.5: Running Time of an **Afl-Step** for Various VP Ratios.

Processor Number :	0	1	2	3	4	5	6	7	8
Active Flag :	1	0	0	1	0	0	0	1	0
Data :	1	4	6	3	1	2	5	2	4
Result of Pack :	1	3	2	X	X	X	X	X	X

Figure 8.6: Example of Pack Operation.

8.4 Taking Advantage of Inactivity

Typically only a small percentage of an AFN is active at any given time. This section briefly outlines a method for taking advantage of this inactivity by only processing the nodes that are active. The method achieves a form of load balancing by spreading the active nodes evenly over all the processors. The method only works when using virtual processors and requires some overhead. It is not clear at what percentage of active nodes the method becomes practical to use.

The method uses an operation called a **pack**, which takes all the active elements and packs them down to the bottom part of memory. Figure 1.6 show the result of a pack operation. Packs run in $O(\log n)$ time on a complete butterfly or hypercube [Batcher74] and run in $O(r \times (\log r + \log n))$ for a VP ratio of r. The CM does not have a complete butterfly across the machine but it does have one within each chip. A chip consists of 16 processors.

To only process the active nodes, we can pack the nodes down to the lowest rows of virtual processors (VPs) so that every processor in the machine has approximately the same number of active VPs. Now when the CM loops over the VPs, it only has to loop over a small percentage to catch all the active ones. Figure 1.7 gives an example of such a packing. Since there is only a butterfly per chip, we usually only pack within the chips. Packing off the chip is significantly more expensive but can be done when the load on different chips becomes significantly unbalanced.

The majority of the time taken by an **afl-step** is spent executing the routing cycle. The following method only packs before routing; it might be worthwhile packing at other times but the gains will be less significant. The method uses basically the same

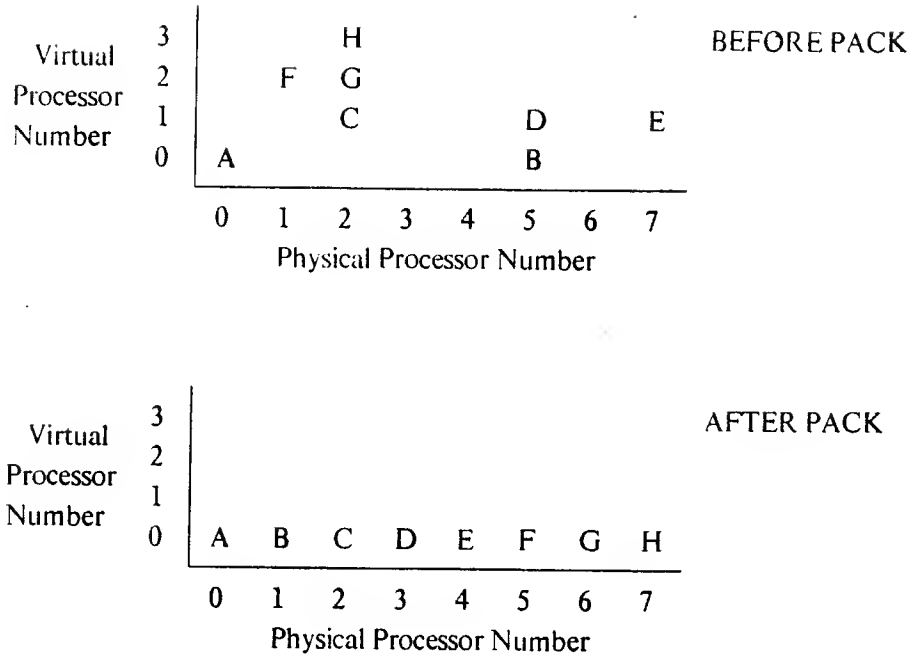


Figure 8.7: Packing on Chip to the Lower VP Rows.

algorithm as described in section 1.3, but before the send instruction (after the link function), all the links with the same value as in the previous **afl-step** are deactivated. The remaining active links append their value to the destination address required by the send cycle and pack them across the processors. The message cycle will now only loop over a subset of the VP banks. If the percentage of active processors is low, this method could save a significant amount of time.

The method has the drawback that it is no longer possible to compile the connections into the router since the source of each message will change between **afl-steps**. No experimentation has been done, so it is not clear where the trade off lies. There are many variations to the method outlined and it is useful for a much more general class of algorithms than discussed in this thesis.

8.5 Counting Links or Nodes

Both the implementations of the **network-processor** discussed in this chapter, and all other implementations I have tried on the CM, have had costs proportional to the number of links rather than the number of nodes in a network. Is this an artifact of the CM? Most likely not - the reason behind the cost of links has a strong theoretical foundation that is independent of any particular machine. This is true even if the links are completely passive (do not require a multiplication).

As was first notice in the field of 2-dimensional VLSI layout [Thompson79, Leiserson80, Lipton81, Valiant81], and later extended to the simulation of arbitrary networks on a fixed network [Bhatt84, Leiserson85], the cost of the processing elements is insignificant as compared to the cost of communications for all but the a few trivial networks, such as trees and 3-d meshes. Cost is measured as a function of the area or volume of the circuit required to implement or simulate a network, and the time taken to run it.

In particular several theorems were derived that give lower bounds on the area or volume required to simulate various networks. These theorems show that for all but trivial networks these bounds are greater than linear in thè the number of nodes, and that for many networks, such as hypercubes, are proportional to the square of the number of nodes. These bounds are based on the assumption that the amount of information that can cross a boundary is proportional to the area of that boundary and do not depend on any properties particular to VLSI layout.

As concerns simulating activity flow networks, two important observations can be abstracted from this work: links are expensive and the locality of nodes is important. These two observations have been neglected by many researchers when analyzing the cost of various *connectionist networks.*

Feldman [81] discusses a method for implementing a dynamic binding scheme for connectionist networks that requires $O(n^{3/2})$ instead of the $O(n^2)$ required by a more obvious scheme. If he had counted links instead of the nodes, he would have found that his method is no cheaper than the simple $O(n^2)$ method.

Touretzky and Hinton [Touretzky85] show how a distributed implementation of a production system on a connectionist network is much cheaper than a *grandfather cell* (local) implementation. Their calculations only consider nodes and ignore links and issues of locality. Each node in their network requires considerably more links than needed in a local scheme, and the links are made randomly and are therefore much harder to collect into local groups needed for local communications. With these considerations, it is not clear that the distributed implementation is cheaper.

When programming with AFL-1 it is easy for the compiler to take advantage of locality for two reasons. Firstly, the use of **node groups** collects nodes into local areas. In general each **node group** communicates more within itself than to any other **node group**. Secondly, many of the predefined abstractions supplied by AFL-1, such as the **fan-in**, **fan-out**, and **mutual-interaction** groups defined in section 4.5, are implementable on commutative trees. Commutative trees are particularly cheap to implement (combining non commutative trees can be expensive as shown by [Lipton81]).

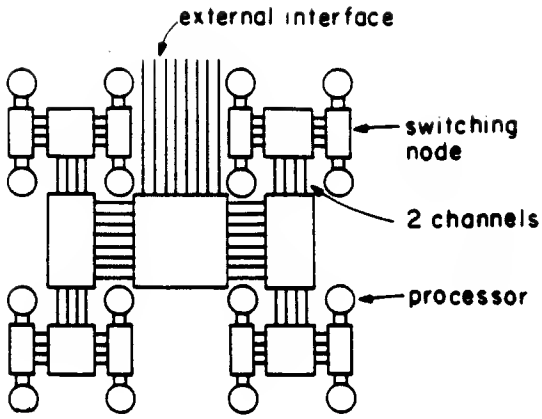


Figure 8.8: A Fat-Tree [Leiserson85].

This is not posed as evidence that local representations are cheaper to implement but rather it suggest that one should be careful about the distribution of connections. It might be reasonable to have distributed representations in which single concepts (concept is intentionally left vague) are distributed over some small volume rather than over the whole network.

8.6 Improvements to the Hardware

The Connection Machine is good for experimenting with AFLs but is more general and therefore costly (time, space and price) than needed. A major problem with the current design is that it is hard to take advantage of locality: in the CM, sending a message 42 processors away costs approximately the same as sending one 50,000 processors away. As mentioned in the previous section, it should be considerably cheaper to implement networks with locality then those without.

The source of the problem in the CM is that it uses a connection topology, a hypercube, that is more powerful and costly than needed. It is therefore impractical to build networks with a large number of processors (by large we mean > 1,000,000). An n node hypercube has the property that whichever way you cut it in half, $n/2$ wires will cross that cut. This requires an extremely high bandwidth. To limit the cost of the CM, the CM only has a hypercube node per chip (16 processors). Because of this, the router must be multiplexed 16 ways.

Better networks for activity flow networks are the so called volume universal networks [Leiserson85], such as *fat-trees* [Leiserson85, Greenberg86]. Figure 1.8 shows a diagram of a fat-tree routing network. A fat-tree resembles a regular binary tree except that the bandwidth increases as you go up the tree. To be volume universal, the

bandwidth must increase by a factor of $2^{2/3}$ at each level.

Another change that would allow faster execution of AFNs is for the hardware to take better advantage of static routing. The CM is optimized for dynamic routing and does not run static routing considerably faster. If a machine is optimized for static routing it is possible to greatly reduce the propagation delay of a routing cycle.

If implemented, these changes coupled with advances in technology are likely to speed up the running time of AFNs by two or three orders of magnitude in the next 10 years. This would make it practical to run networks with over a billion links.

Chapter 9

Conclusion

A language was presented that allows the hierarchical description of networks of very simple computational devices. The language, AFL-1, is similar to many circuit design languages, but unlike those languages, the networks AFL-1 creates are designed to be simulated not etched into silicon. Because the primitive network elements are simple and only a few different types exist, it is cheap and easy to simulate the networks on massively concurrent SIMD machines such as the Connection Machine.

Two tasks were programmed in the language: a rule based system (CIS), and a planning system (AFPLAN). An analysis of the CIS showed that much of the same functionality can be achieved using the network model and the AFL-1 language as is found in existing rule based systems such as MYCIN. Because the networks are massively concurrent, CIS has the potential of simulating many more rules in real time applications than serial production systems do.

The abstraction supplied by circuit computation languages such as AFL-1 are very different form those found in conventional languages. Because of this, the approach the programmer takes toward achieving a certain functionality will differ substantially. One important abstraction of AFL-1 is the mutually exclusive group (ME-group). Several flavors of this group were discussed and an efficient implementation of the groups was given. CIS and AFPLAN both use such groups and it is likely that most systems could make significant use of them.

To understand which structures are cheap and which are expensive, one has to study the properties of the simulators used to run the networks. A discussion of the AFL-1 network simulator was given. Two important aspects of simulating networks were discussed. These were that links are at least as expensive as nodes, and that it is important to collect nodes into local groups.

This thesis introduces the idea of hierarchical programming languages for circuit like models of computation. Much work needs to be done before the potential of such languages is well understood.

Bibliography

Ackley, D.H., Hinton, G.E., Sejnowski, T.J., "A Learning Algorithm for boltzmann Machines", Cognitive Science, 1985, 9, 147-169.

Agre, P.E., "Routines", Memo 828, MIT AI Laboratory, Many 1985.

Ballard, D.H., Hayes, P.J., "Parallel Logical Inference", Conference of the Cognitive Science Society, 1984, Boulder, CO, 114-123.

Batali, J., Hartheimer, A., "The Design Procedure Language Manual", Memo 598, MIT AI Laboratory, September 1980.

Batcher, K.E., "STARAN Parallel Processor System Hardware", Proc. AFIPS-NCC, 1974, vol.43, 405-410.

Batcher, K.E., "The Flip Network of Staran", Proc. Int'l. Conf. on Parallel Processing, 1976, Detroit, Mich., 65-71.

Batcher, K.E., "Design of a Massively Parallel Processor", IEEE Trans. on Comp., 1980, C-29, 9, 836-840.

Bawden, A., Agre, P.E., "What a Parallel Programming Language has to Let You Say", Memo 796, MIT AI Laboratory, September 1984.

Bawden, A., "A programming Language for Massively Parallel Computers", MS Thesis, Dept. of Electrical Engineering and Computer Science, MIT, October 1984.

Bhatt, S.N., Leighton, F.T., "A Framework for Solving VLSI Graph Layout Problems" Journal of Computer and System Sciences, April 1984, Vol. 28, No.2, 300-343.

Bouknight, W.J., Denenberg, S.A., McIntyre, D.E., Randall, J.M., Sameh, A.H., Slotnick, D.L., "The Illiac IV System", Proc IEEE, April 1972, vol. 60, no. 4, 369-388.

Chapman, D., "Planning for Conjunctive Goals", TR-802, MIT AI Laboratory, November 1985.

Christman, D.P., "Programming the Connection Machine", MS Thesis, Dept. of Electrical Engineering and Computer Science, MIT, January 1984.

Collins, A.M., Quillian, M.R., "Experiments on Semantic Memory and Language Com-

prehension", in L.W. Greg (Ed.), *Cognition in Learning and Memory*, New York, Wiley, 1972.

Collins, A.M., Loftus, E.F., "A Spreading Activation Theory of Semantic Processing", Psychological Review, 1975, 82, 407-428.

Cottrell, G.W., Small, S.L., "A Connectionist Scheme for Modeling Word Sense Disambiguation", Cognition and Brain Theory, 1983, 6, 89-120.

Cottrell, G.W., "Connectionist Parsing", Conference of the Cognitive Science Society, 1985, Irvine, CA, 201-211.

Cottrell, G.W., "A Connectionist Approach to Word Sense Disambiguation", TR 154, Computer Science Department, University of Rochester, May 1985.

Crowther, W., Goodhue, J., Starr, E., Thomas. R., Milliken, W., Bleckadar, T., "Performance Measurements on a 128-Node Butterfly Parallel Processor", Proc. Int'l. Conf. Parallel Processing, August 1985, 531-540.

Davis, A.L., Robison, S.V., "The Architecture of the FAIM-1 Symbolic Multiprocessing System", Proc IJCAI, August 1985, Los Angeles, 32-38.

Davis, Randall, Buchanan, B., Shortliffe, E., "Production Rules as a Representation for Knowledge-Based Consultation Program", Artificial Intelligence, 1977, 8, 15-45.

Davis, Randall, "Meta-Rules: Reasoning about Control", Artificial Intelligence, 1980, 15, 179-222.

Davis, Ronald, Thomas, D., "Systolic Array Chip Matches the Pace of High-Speed Processing" Electronic Design, October 1984.

Denneau, M.M., "The Yorktown Simulation Engine", Proc. of the 19th Design Automation Conf., June 1982, 55-59.

Deutsch, J.T., Newton, A.R., "MSPLICE: A Multiprocessor-Based Circuit Simulator", Proc. Int'l Conf. on Parallel Processing, August 1984, 207-214.

Douglass, R.J., "A Qualitative Assessment of Parallelism in Expert Systems", IEEE Software, May 1985, 70-81.

Doyle, J., "A Truth Maintenance System", Artificial Intelligence, 1979, Vol. 12, No. 3.

Duda, R.O., Hart, P.E., and Nilson, N.J., "Subjective Bayesian Methods for Rule-Based Inference Systems", Proc. National Computer Conf, 1976, 45, 1075-1082.

Etherington, D.W., Reiter, R., "On Inheritance Hierarchies With Exceptions", Proc. AAAI, August 1983, Washington D.C., 104-108.

Fahlman, S.E., *NETL: A System For Representing and Using Real-World Knowledge*, Cambridge, Mass., MIT Press, 1979.

Fahlman, S.E., Touretzky, D.S., van Roggen, W., "Cancellation in a Parallel Semantic Network", Proc. IJCAI, August 1981, Vancouver, 257-263.

Fahlman, S.E., "Three Flavors of Parallelism", Proc. National Conference of the Canadian Society for Computational Studies of Intelligence, May 1982, Saskatoon, Saskatchewan, 230-235.

Fahlman, S.E., Hinton G.E., Sejnowski, T.J., "Massively Parallel Architectures for AI: NETL, THISTLE and Boltzmann Machines", Proc. AAAI, August 1983, Washington D.C, 109-113.

Farmwald, P.M., "The S-1 Mark IIA Supercomputer", in J.S. Kowalik (Ed.), *High-Speed Computation*, New York, Springer-Verlag, 1984.

Feldman, J.A., "A Connectionist Model of Visual Memory", in G.E. Hinton and J.A. Anderson (Eds.), *Parallel Models of Associative Memory*, Hillsdale, NJ, Erlbaum, 1981.

Feldman, J.A., "Dynamic Connections in Neural Networks", Biological Cybernetics, 1982, 46, 27-39.

Feldman, J.A., Ballard, D.H., "Connectionist Models and Their Properties", Cognitive Science, 1982, 6, 205-254.

Feldman, J.A., Shastri, L., "Evidential Inference in Activation Networks", Conference of the Cognitive Science Society, June 1984, Boulder, CA, 156-160.

Feldman, J.A., "Four Frames Suffice: A Provisional Model of Vision and Space", The Behavioral and Brain Sciences, 1985, 8, 265-289.

Fikes, R.E., Hart, P.E., Nilsson, N.J., "STRIPS: A New Approach to the Application of Theorem Proving to Problem Solving", Artificial Intelligence, 1971, 2, 198-208.

Fitch, F., "New Bounds for Parallel Prefix Circuits", Oroc. of the 15th Annual ACM Symposium on Theory of Computing, May 1983, 100-109.

Forgy, C.L., "The OPS5 User's Manual", Technical Report, Carnegie-Mellon University, Department of Computer Science, 1981.

Forgy, C., A. Gupta, A. Newell, R. Wedig, "Initial Assessment of Architectures for Production Systems", Proc. AAAI, August 1984, Austin, TX., 116-120.

Fox, M.S., Lowenfeld, S., Kleinosky, P., "Techniques for Sensor-Based Diagnostics" Proc. IJCAI, August 1983, Karlsruhe W. Germany, 158-163.

Gaschnig, J., "Prospector: An Expert System for Mineral Exploration", in Michie (Ed.), *Introductory Readings in Expert Systems,* New York, Gordon and Breach, 1982.

Golberg, A., Robson, D., *Smalltalk-80* Reading, Mass., Addison-Wesley, 1983.

Gottlieb, A., Grishman, R., Kruskal, C.P., McAuliffe, K.P., Rudolph, L., Snir, M., "The NYU Ultracomputer - Designing a MIMD, Shared-Memory Parallel Machine", IEEE Trans. on Computers, 1983, C-32, 175-189.

Greenberg, R.I., Leiserson, C.E., "Randomized Routing on Fat-Trees", Forthcoming.

Gupta, A., Forgy, C., Newell, A., Wedig, R., "Parallel Algorithms and Architectures for Rule-Based Systems", Proc. Int'l. Symposium on Comp. Arch., June 1986.

Hayes, J.P., "A Unified Switching Theory with Applications to VLSI Design", Proc. IEEE, 1982, 70, 10, 1140-1151.

Hewitt, C., "Procedural Embedding of Knowledge in Planner", Proc. IJCAI, 1971, 167-182.

Hewitt, C., Lieberman, H., "Design Issues in Parallel Architectures for Artificial Intelligence", Memo 750, MIT AI Laboratory, November 1983.

Hillis, W. D., "The Connection Machine (Computer Architecture for the New Wave)", Memo 646, MIT AI Laboratory, September 1981.

Hillis, W. D., *The Connection Machine*, Cambridge, Mass., MIT Press, 1985.

Hinton, G.E., *Relaxation and its Role in Vision*, University of Edinburgh: Doctoral Dissertation, December 1977.

Hinton, G.E., "Shape Representation in Parallel Systems", Proc. IJCAI, August 1981, Vancouver, 1088-1096.

Hinton, G.E., "Implementing Semantic Networks in Parallel", in G.E. Hinton and J.A. Anderson (Ed.), *Parallel Models of Associative Memory*, Hillsdale, NJ: Erlbaum, 1981.

Hinton, G.E., Sejnowski, T.J., "Analyzing Cooperative Computation", Proc. of the Fifth Annual Conference of the Cognitive Science Society, May 1983, Rochester NY, Session 7.

Hinton, G.E., Lang, K.J., "Shape Recognition and Illusory Conjunctions", Proc. IJCAI, 1985, Los Angeles, 252-259.

Hopfield, J.J., "Neural networks and physical systems with emergent collective computational abilities", Proc. National Academy of Sciences USA, 1982, 79, 2554-2558.

Kirpatrick, S., Gelatt, C.D., Vecchi, M.P., "Optimization by Simulated Annealing", Science, 1983, 220, 671-680.

Koch, C., Ullman, S., "Selecting One Among the Many: A Simple Network Implementing Shifts in Selective Visual Attention", Memo 770, MIT AI Laboratory, January 1984.

Kruskal, C.P., Rudolph, L., Snir, M., "The Power of Parallel Prefix", Proc. Int'l. Conference on Parallel Processing, August 1985, 180-185.

Kung, H.T., "Systolic Algorithms for the CMU Warp Processor", Proc. 7th Int'l Conf. on Pat. Recognition, July-August 1984, 570-577.

Lasser, C., "The *Lisp Manual", Thinking Machines Corporation, Cambridge, Mass., 1986.

Leiserson, C.E., "Area-Efficient Layouts (for VLSI)", 21st Annual IEEE Symp. on Foundations of Computer Science, 1980.

Leiserson, C.E., "Fat-Trees: Universal Networks for Hardware-Efficient Supercomputing", IEEE Transactions on Computers, October 1985, c-34, 10, 892-901.

Lenat, D.B., "The Ubiquity of Discovery", Artificial Intelligence, 1978, 9, 257-285.

Lewis, P.M., Coates, C.L., *Threshold Logic*, New York, Wiley, 1967.

Lipton, R.J., Sedgewick, R., "Lower Bounds for VLSI", Proc. STOC, May 1981, Milwaukee, 300-3007.

McCulloch, W.S., Pitts, W., "A Logical Calculus of the Ideas Imminent in Nervous Activity", Bulletin of Mathematical Biophysics, Vol. 5, 1943.

McDermott, J., "R1: an Expert in the Computer Systems Domain", Proc. AAAI, August 1980, Stanford University, 269-271.

Miller, R.A., Pople, H.E., Myers, J.D., "Internist-1, An Experimental Computer-Based Diagnostic Consultant for General Internal Medicine", in W. Clancy and E. Shortliffe (Ed.), *Readings in Medical Artificial Intelligence*, Reading, Mass., Addison-Wesley, 1984.

Minsky, M., "K-lines: A Theory of Memory", Cognitive Science, 1980, 4, 117-133.

Minsky, M., "Plain Talk About Neurodevelopmental Epistemology", Proc. IJCAI, August 1977, Cambridge Mass., 1083-1092.

Minsky, M., Papert, S., *Perceptrons*, Cambridge, Mass., MIT Press, 1969.

Minsky, M., *Society of Minds*, Forthcoming.

Murakami, K., Kakuta, T., Onai, R., "Architectures and Hardware Systems: Parallel Inference Machine and Knowledge Base Machine", Proc. Int'l Conf. Fifth Generation Computer Systems, 1984, Tokyo, 18-36.

Newell, A., Simon, H.A., *Human Problem Solving*, Englewood Cliffs, NJ: Prentice Hall, 1972.

Oflazer, K., "Partitioning in Parallel Processing of Production Systems", Proc. Int'l Conf. Parallel Processing, August 1984, 92-100.

Quillian, M.R., "Semantic Memory", in Minsky (ed.), *Semantic Information Processing*, Cambridge, Mass., MIT Press, 1968.

Reggia, J.A., "Virtual Lateral Inhibition In Parallel Activation Models of Associative Memory", Proc. IJCAI, August 1985, Los Angeles, 244-248.

Rosenblatt, F., "A Comparison of Several Perceptron Models", Proceedings of a Symposium on Mechanization of Thought Processes, 1958.

Rosenblatt, F., *Principals of Neurodynamics: Perceptrons and the Theory of Brain Mechanics.* Washington D.C., Spartan, 1961.

Rumelhart, D.E., Zipser, D., "Feature Discovery by Competitive Learning", Cognitive Science, 1985, 9, 75-112.

Rumelhart, D.E., McClelland, J.L., *Parallel Distributed Processing: Explorations in the Microstructure of Cognition, Volume 1: Foundations*, MIT Press, Cambridge Mass., 1986.

Sacerdoti, E.D., "The Nonlinear Nature of Plans", Advance Papers of the 4th IJCAI,

August 1975, Tbilisi, USSR. 206-214.

Schwartz, J.T., "Ultracomputers", ACM Transactions on Programming Languages and Systems, Vol. 2, No. 4, October 1980, pages 484-521.

Selman, B., Hirst, G., "A Rule-Based Connectionist Parsing System", Conference of the Cognitive Science Society, August 1985, Irvine, CA, 212-221.

Shastri, L., Feldman, J.A., "Evidential Reasoning in Semantic Networks: A Formal Theory", Proc. IJCAI, August 1985, Los Angeles, 465-474.

Shastri, L., "Evidential Reasoning in Semantic Networks: A Formal Theory and its Parallel Implementation", TR 166, Computer Science Department, University of Rochester, September 1985.

Shaw, D.E., "NON-VON: A Parallel Machine Architecture for Knowledge-Based Information Processing", Proc. IJCAI, August 1981, Vancouver, 961-963.

Shaw, D.E., "Organization and Operation of a Massively Parallel Machine", in Guy Rabbat (ed.), *Computers and Technology*, Elsevier-North Holland, 1985.

Shortliffe, E. H., Buchanan, B. G., "A Model of Inexact Reasoning in Medicine", Math. Biosci., 1975, 23, 351-379.

Siewiorek, D.P., Kini, V., Mashburn, H., Joobbani, R., "A Case Study of C.mmp, Cm* and C.vmp, Part II: Predicting and Calibrating Reliability of Multiprocessor Systems", Proc. IEEE, 1978, vol. 66, no. 10, 1,2000-1,220.

Small, S., Cottrell, G., Shastri, L., "Toward Connectionist Parsing", Proc. AAAI, August 1982, Pittsburgh, PA, 247-250.

Smith, B.J., "Latency and HEP", in J.S. Kowalik (Ed.), *High-Speed Computation*, New York, Springer-Verlag, 1984.

Steele, G.L. Jr., "The Definition and Implementation of a Computer Programming Language Based on Constraints", AI-TR 595, MIT AI Laboratory, August 1980.

Steele, G.L. Jr., *Common Lisp*, Burlington, Mass., Digital Press, 1984.

Stefik, M., "Planning with Constraints (MOLGEN: Part 1)", Artificial Intelligence, 1981, 16, 111-140.

Stolfo, S. J., Miranker, D., Shaw, D. E., "Architecture and Applications of DADO: A Large-Scale Parallel Computer For Artificial Intelligence", Proc. IJCAI, August

1983, Karlsruhe W. Germany, 850-854.

Sussman, G.J., "A Computational Model of Skill Acquisition", AI-TR 297, MIT AI Laboratory, August 1973.

Sussman, G.J., Steele, G.L. Jr., "Constraints - A Language for Expressing Almost-Hierarchical Descriptions", Artificial Intelligence, 1980, 14, 1-39.

Swartout, William R., "Explaining and Justifying Expert Consulting Programs", Proc IJCAI, August 1981, Vancouver, 815-823.

Symbolics Inc., "Reference Guide to Symbolics Lisp", Cambridge, Mass., 1985.

Symbolics Inc., "User's Guide to Symbolics Computers" Cambridge, Mass., 1985.

Thompson, C.D., "Area-Time Complexity for VLSI", Proc. STOC, May 1979, Atlanta, Georgia, 81-88.

Thurber, K.J., *Large Scale Computer Architecture - Parallel and Associative Processors*, N.J., Hayden Book Co., 1976.

Touretzky, D.S., "Symbols Among the Neurons: Details of a Connectionist Inference Architecture", Proc. IJCAI, August 1985, Los Angeles, 238-243.

Treisman, A.M., Shmidt, H., "Illusory Conjunctions in the Perception of Objects", Cognitive Psychology, 1982, 14, 107-141.

Uttley, A.M., "Conditional Probability Computing in a Nervous System", Second Symposium on Self Organizing Systems, 1962.

Valiant, L.G., "Universality Considerations in VLSI Circuits", IEEE Transactions on Computers, February 1981, c-30, 2, 135-140.

Vesonder, G.T., Stolfo, S.J., Zielinski, J.E., Miller, F.D., Copp, D.H., "ACE: an Expert System for Telephone Cable Maintenance", Proc. IJCAI, August 1983, Karlsruhe W. Germany, 116-121.

Waltz, D.L., Pollack, J.B., "Massively Parallel Parsing: A Strongly Interactive Model of Natural Language Interpretation", Cognitive Science, 1985, 9, 51-74.

Woods, W.A., "Research in Natural Language Understanding, Progress Report No. 2", Report No. 3797, Bolt Beranek and Newman Inc., Cambridge, MA, April 1978.

Zadeh, L.A., "Fuzzy Sets", Information and Control, 1965, 8, 338-353.

CS-TR Scanning Project
Document Control Form

Date : 10 / 18 / 95

Report # AI-TR-918

Each of the following should be identified by a checkmark:
Originating Department:

☒ Artificial Intellegence Laboratory (AI)
☐ Laboratory for Computer Science (LCS)

Document Type:

☒ Technical Report (TR) ☐ Technical Memo (TM)
☐ Other:_____

Document Information

Number of pages: 139 (139 - IMAGES)
Not to include DOD forms, printer intstructions, etc... original pages only.

Originals are:
☐ Single-sided or
☒ Double-sided

Intended to be printed as :
☐ Single-sided or
☒ Double-sided

Print type:
☐ Typewriter ☐ Offset Press ☒ Laser Print
☐ InkJet Printer ☐ Unknown ☐ Other:_____

Check each if included with document:

☒ DOD Form (2) ☐ Funding Agent Form ☒ Cover Page
☐ Spine ☐ Printers Notes ☐ Photo negatives
☐ Other: _____

Page Data:

Blank Pages(by page number): FOLLOWS TITLE PAGE

Photographs/Tonal Material (by page number):_____

Other (note description/page number):

Description :	Page Number:
IMAGE MAP: (1-139) UNNUMBERED TITLE & BLANK PAGES	3-132
(133-139) SCAN CONTROL, COVER, DOD (2), TRGTS (3)	

Scanning Agent Signoff:

Date Received: 10/18/95 Date Scanned: 10/20/95 Date Returned: 10/26/95

Scanning Agent Signature: Michael N. Cook

UNCLASSIFIED

SECURITY CLASSIFICATION OF THIS PAGE (When Data Entered)

REPORT DOCUMENTATION PAGE

READ INSTRUCTIONS BEFORE COMPLETING FORM

1. REPORT NUMBER: AI-TR-918	2. GOVT ACCESSION NO.
3. RECIPIENT'S CATALOG NUMBER	
4. TITLE (and Subtitle): AFL-1: A Programming Language for Massively Concurrent Computers	5. TYPE OF REPORT & PERIOD COVERED: Technical Report
6. PERFORMING ORG. REPORT NUMBER	
7. AUTHOR(s): Guy E. Blelloch	8. CONTRACT OR GRANT NUMBER(s): N00014-85-K-0124
9. PERFORMING ORGANIZATION NAME AND ADDRESS: Artificial Intelligence Laboratory, 545 Technology Square, Cambridge, MA 02139	10. PROGRAM ELEMENT, PROJECT, TASK AREA & WORK UNIT NUMBERS
11. CONTROLLING OFFICE NAME AND ADDRESS: Advanced Research Projects Agency, 1400 Wilson Blvd., Arlington, VA 22209	12. REPORT DATE: November 1986 13. NUMBER OF PAGES: 132
14. MONITORING AGENCY NAME & ADDRESS (if different from Controlling Office): Office of Naval Research, Information Systems, Arlington, VA 22217	15. SECURITY CLASS. (of this report): UNCLASSIFIED 15a. DECLASSIFICATION/DOWNGRADING SCHEDULE

16. DISTRIBUTION STATEMENT (of this Report)

Distribution is unlimited.

17. DISTRIBUTION STATEMENT (of the abstract entered in Block 20, if different from Report)

18. SUPPLEMENTARY NOTES

None

19. KEY WORDS (Continue on reverse side if necessary and identify by block number)

Programming Languages, Massively Parallel Systems, Connectionist Networks, Activity Flow, Connection Machine, Rule Based Systems.

20. ABSTRACT (Continue on reverse side if necessary and identify by block number)

Computational models are arising in which programs are constructed by specifying large networks of very simple computational devices. Although such models can potentially make use of a massive amount of concurrency, their usefulness as a programming model for the design of complex systems will ultimately be decided by the ease in which such networks can be programmed (constructed). This report outlines a language for specifying computational networks. The language (AFL-1) consists of a set of primitives,

DD FORM 1 JAN 73 1473 EDITION OF 1 NOV 65 IS OBSOLETE S/N 0102-014-6601

UNCLASSIFIED

SECURITY CLASSIFICATION OF THIS PAGE (When Data Entered)

Block 20 Continued

and a mechanism to group these elements into higher level structures. An implementation of this language runs on the Thinking Machines Corporation Connection Machine. Two significant examples were programmed in the language, an expert system (CIS), and a planning system (AFPLAN). These systems are explained and analyzed in terms of how they compare with similar systems written in conventional languages.

Scanning Agent Identification Target

Scanning of this document was supported in part by the **Corporation for National Research Initiatives,** using funds from the **Advanced Research Projects Agency** of the **United states Government** under Grant: **MDA972-92-J1029.**

The scanning agent for this project was the **Document Services** department of the **M.I.T Libraries.** Technical support for this project was also provided by the **M.I.T. Laboratory for Computer Sciences.**